

CONTEXT

MKII

MKIV

CONTEXT



# Contents

Introduction	3
I From MkII to MkIV	5
II How Lua fits in	7
III Initialization revised	19
IV An example: CalcMath	23
V Going utf	27
VI A fresh look at fonts	31
VII Token speak	47
VIII How about performance	57
IX Nodes and attributes	65
X Dirty tricks	75
XI Going beta	79
XII Zapfing fonts	87
XIII Arabic	99
XIV Colors redone	105
XV Chinese, Japanese and Korean, aka CJK	115
XVI Optimization	121
XVII XML revisioned	127
XVIII Breaking apart	141
XIX Collecting garbage	147



# 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 spend 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. Keep tuned,

Hans Hagen, Hasselt NL,  
August 2006 and beyond

<http://www.luatex.org>



# I From MkII to MkIV

Sometime in 2005 the development of L<sup>A</sup>T<sub>E</sub>X started, a further development of P<sup>D</sup>F<sub>T</sub><sub>E</sub>X and a precursor to P<sup>D</sup>F<sub>T</sub><sub>E</sub>X version 2. This T<sub>E</sub>X variant will provide:

- 21--32 bit internals plus a code cleanup
- flexible support for O<sup>P</sup>E<sup>N</sup>T<sup>Y</sup>P<sup>E</sup> fonts
- an internal U<sup>T</sup>F data flow
- the bidirectional typesetting of A<sup>L</sup>E<sup>P</sup>H
- L<sup>U</sup>A callbacks to the most relevant T<sub>E</sub>X internals
- some extensions to T<sub>E</sub>X (for instance math)
- an efficient way to communicate with M<sup>E</sup>T<sup>A</sup>P<sup>O</sup>ST

In the tradition of T<sub>E</sub>X this successor will be downward compatible in most essential parts and in the end, there is still P<sup>D</sup>F<sub>T</sub><sub>E</sub>X version 1 as fall back.

In the mean time we have seen another unicode variant show up, X<sub>Y</sub>T<sub>E</sub>X which is under active development, uses external libraries, provides access to the fonts on the operating system, etc.

From the beginning, C<sup>O</sup>N<sup>T</sup><sub>E</sub>X<sup>T</sup> 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  $\epsilon$ -T<sub>E</sub>X, A<sup>L</sup>E<sup>P</sup>H, and X<sub>Y</sub>T<sub>E</sub>X. Because nowadays all engines provide the  $\epsilon$ -T<sub>E</sub>X features, in August 2006 we decided to consider those features to be present and drop providing the standard T<sub>E</sub>X compatible variants. This is a small effort because all code that is sensitive for optimization already has  $\epsilon$ -T<sub>E</sub>X code branches for many years.

However, with the arrival of L<sup>A</sup>T<sub>E</sub>X, we need a more drastic approach. Quite some existing code can go away and will be replaced by different solutions. Where T<sub>E</sub>X code ends up in the format file, along with its state, L<sup>U</sup>A code will be initiated at run time, after a L<sup>U</sup>A instance is started. C<sup>O</sup>N<sup>T</sup><sub>E</sub>X<sup>T</sup> reserves its own instance of L<sup>U</sup>A.

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 C<sup>O</sup>N<sup>T</sup><sub>E</sub>X<sup>T</sup>'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.

- **CON<sub>T</sub>EX<sub>T</sub> MkI** At that moment we still had a low level Dutch interface, invisible for users but not for developers.
- **CON<sub>T</sub>EX<sub>T</sub> MkII** We now have a low level English interface, which (as we indeed saw happen) triggers more development by users.
- **CON<sub>T</sub>EX<sub>T</sub> MkIV** This is the next generation of CON<sub>T</sub>EX<sub>T</sub>, 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 X<sub>Y</sub>TEX is stable, we may use that slot for X<sub>Y</sub>TEX 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 T<sub>E</sub>X ( $\epsilon$ -T<sub>E</sub>X, PDF<sub>T</sub>E<sub>X</sub>, A<sub>L</sub>E<sub>P</sub>H).
- **somefile.mkiv**: code optimized for use with L<sub>U</sub>A<sub>T</sub>E<sub>X</sub>, which could follow completely different approaches
- **somefile.lua**: L<sub>U</sub>A 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 transparently replaces `MkII` code and users will seldom notice 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 run-time 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 re-organize 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<sup>+</sup> is that we will have an alternative output mode. In the perspective of L<sup>A</sup>T<sub>E</sub>X it makes sense to have a L<sup>A</sup> 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 L<sup>A</sup> than in T<sub>E</sub>X.

The MkIV converter is a multi-pass converter. First we clean up the METAPost output, next we convert the POSTSCRIPT code into L<sup>A</sup> calls. We assume that this L<sup>A</sup> code eventually can be output directly from METAPost. We then evaluate this converted L<sup>A</sup> blob, which results in T<sub>E</sub>X commands. Think of:

1.2 setlinejoin

turned into:

mp.setlinejoin(1.2)

becoming:

\PDFcode{1.2 j}

which is, when the PDF<sub>T</sub>E<sub>X</sub> 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 L<sup>A</sup> machinery.

When the MkIV converter reached a stable state, tests demonstrated then the code was upto 20% slower than the pure T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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.

Cleaning up the METAPost output using L<sup>A</sup> 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

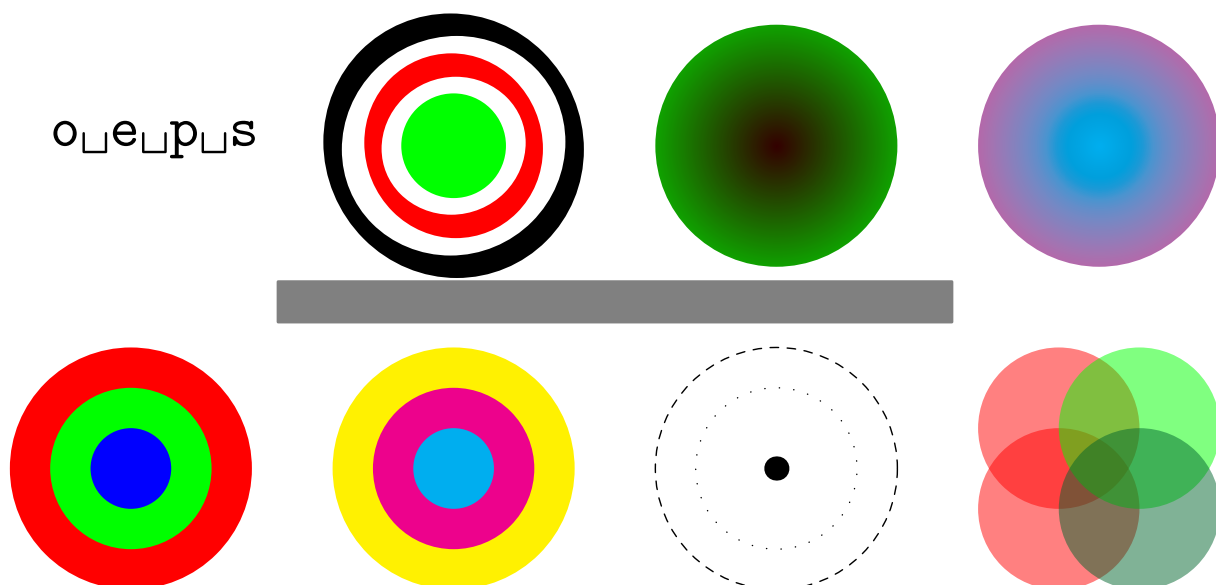


Figure II.1 converter test figure

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 rigorous 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 T<sub>E</sub>X commands, we need only 0.3 seconds for embedding the 250 instances. This includes T<sub>E</sub>X 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 T<sub>E</sub>X 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 CON<sub>T</sub>E<sub>X</sub>T 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 T<sub>E</sub>X job handle the typesetting of texts. A side effect of these experiments was a new mechanism in CON<sub>T</sub>E<sub>X</sub>T (and METAFUN) where T<sub>E</sub>X did all typesetting of labels, and METAPost only worked with an abstract representation of the result. This way we can completely avoid nested T<sub>E</sub>X 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), took mere seconds compared to minutes when using independent METAPost runs for each job.

## characters

Because L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X was already a merge between P<sub>D</sub>F<sub>T</sub>E<sub>X</sub> 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 T<sub>E</sub>X is larger than that of good old T<sub>E</sub>X, even with some of the clever memory allocation techniques as used in L<sup>A</sup>T<sub>E</sub>X. 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.

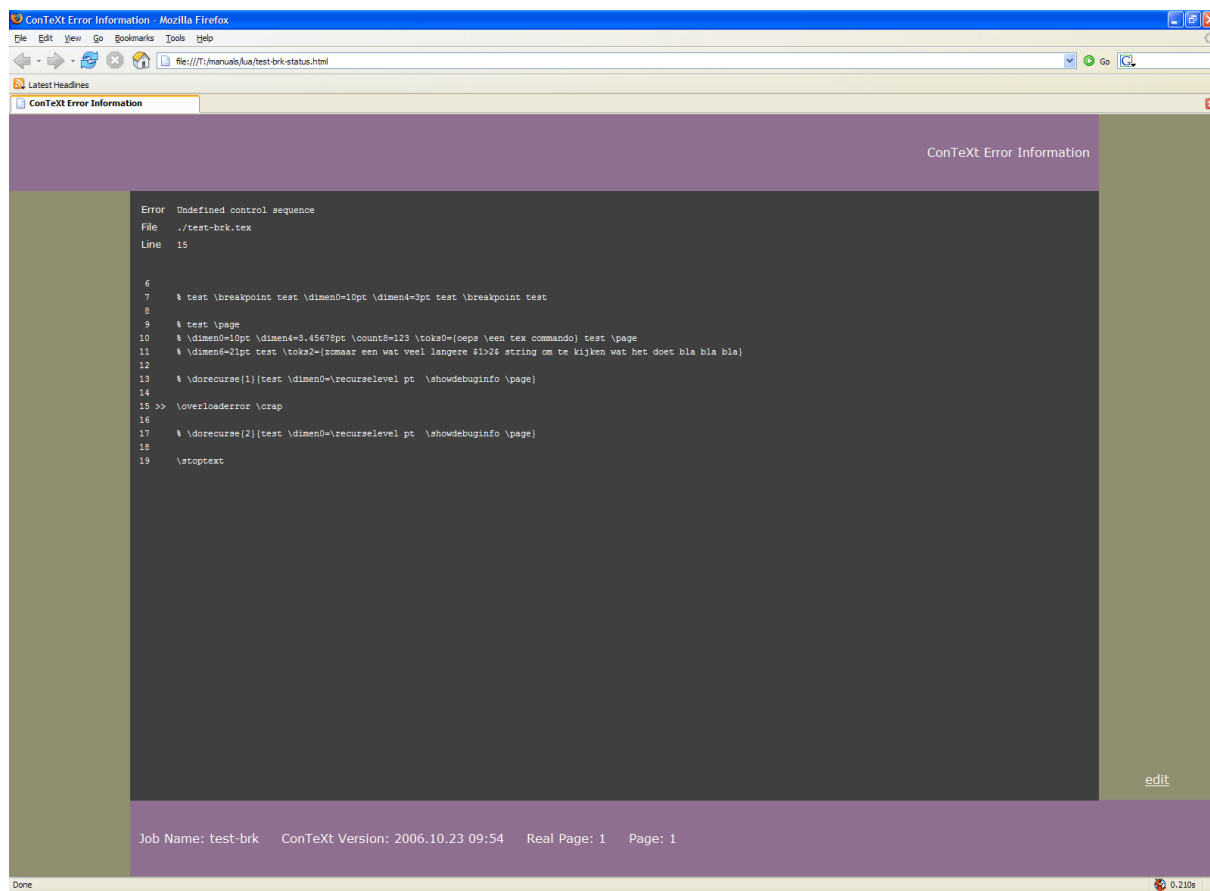
*How stable are the mentioned numbers? Ten months after writing the previous text we get the following numbers:*

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, L<sup>A</sup>T<sub>E</sub>X's format has become much larger! Partly this is the result of more LUA code, especially O<sub>P</sub>E<sub>N</sub>T<sub>Y</sub>P<sub>E</sub> font handling and attributes related code. The extra T<sub>E</sub>X 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 T<sub>E</sub>X 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.II and figure II.III you can see an example of this.



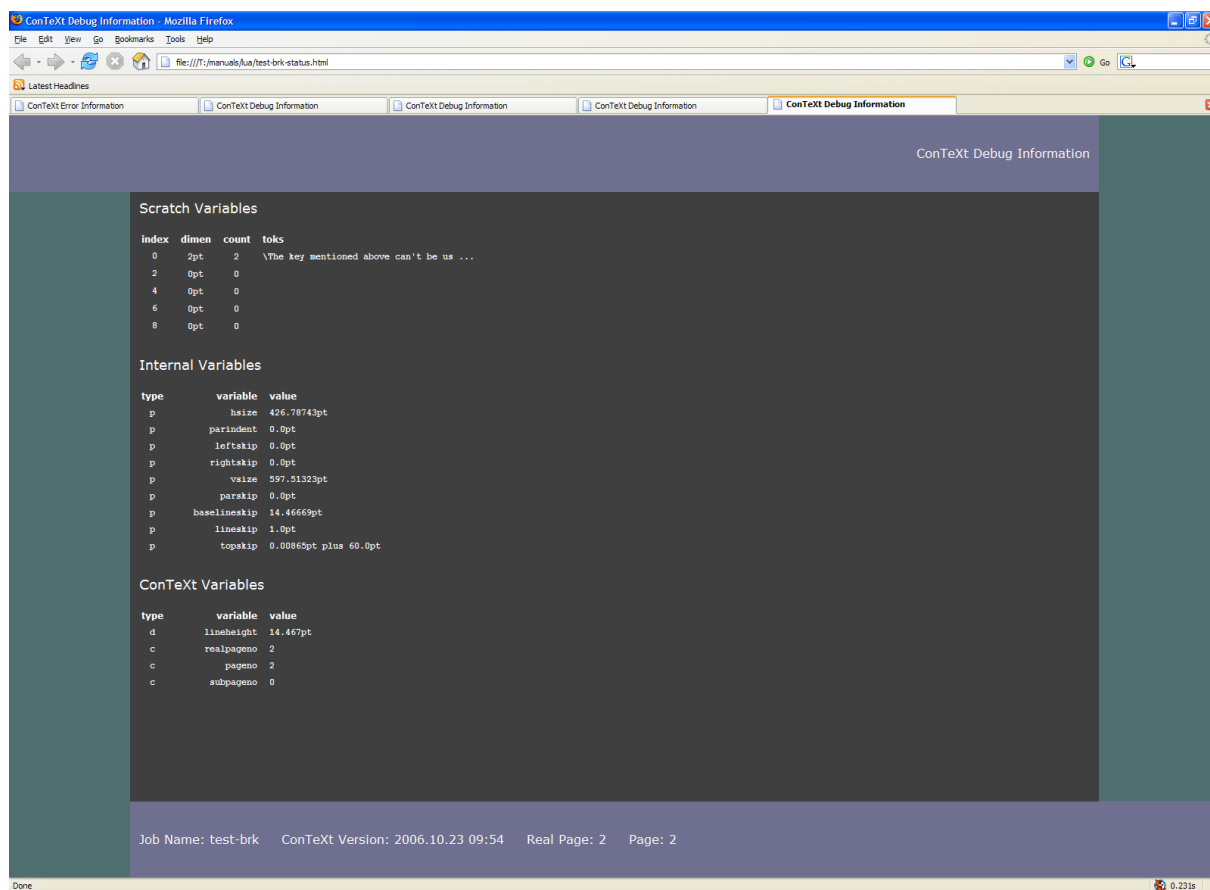
**Figure II.II** An example error screen.

Playing with such features gives us an impression of what kind of access we need to T<sub>E</sub>X's internals. It also formed a starting point for conversion routines and a mechanism for embedding LUA code in HTML pages generated by CON<sub>T</sub>E<sub>X</sub>T.

## file io

Replacing T<sub>E</sub>X's in- and output handling is non-trivial. 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 T<sub>E</sub>X and LUA code was thrown



**Figure II.III** An example debug screen.

away as soon as better solutions came around. Because we were testing on real documents, using a fully loaded CONTEX<sub>T</sub> 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 L<sup>A</sup>T<sub>E</sub>X. If you want to overload the file searching and IO handling, you need overload as soon as possible. Because L<sup>A</sup>T<sub>E</sub>X is also supposed to work with the existing K<sub>P</sub>S<sub>E</sub> library, we still have that as fallback, but in principle one could think of a K<sub>P</sub>S<sub>E</sub> 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 source code has K<sub>P</sub>S<sub>E</sub> calls and references to K<sub>P</sub>S<sub>E</sub> 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 R<sub>U</sub>B<sub>Y</sub> based K<sub>P</sub>S<sub>E</sub> variant into L<sub>U</sub>A and started working from that point. The advantage of having our own IO handler is that we can go beyond K<sub>P</sub>S<sub>E</sub>. For instance, since L<sup>A</sup>T<sub>E</sub>X has, among a few others, the ZIP libraries linked in, we can read from ZIP files, and keep all T<sub>E</sub>X related files in TDS compliant ZIP files as well. This means that one can say:

```
\input zip::somezipfile::somefile.tex
\input zip://somezipfile.zip/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 beyond 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).



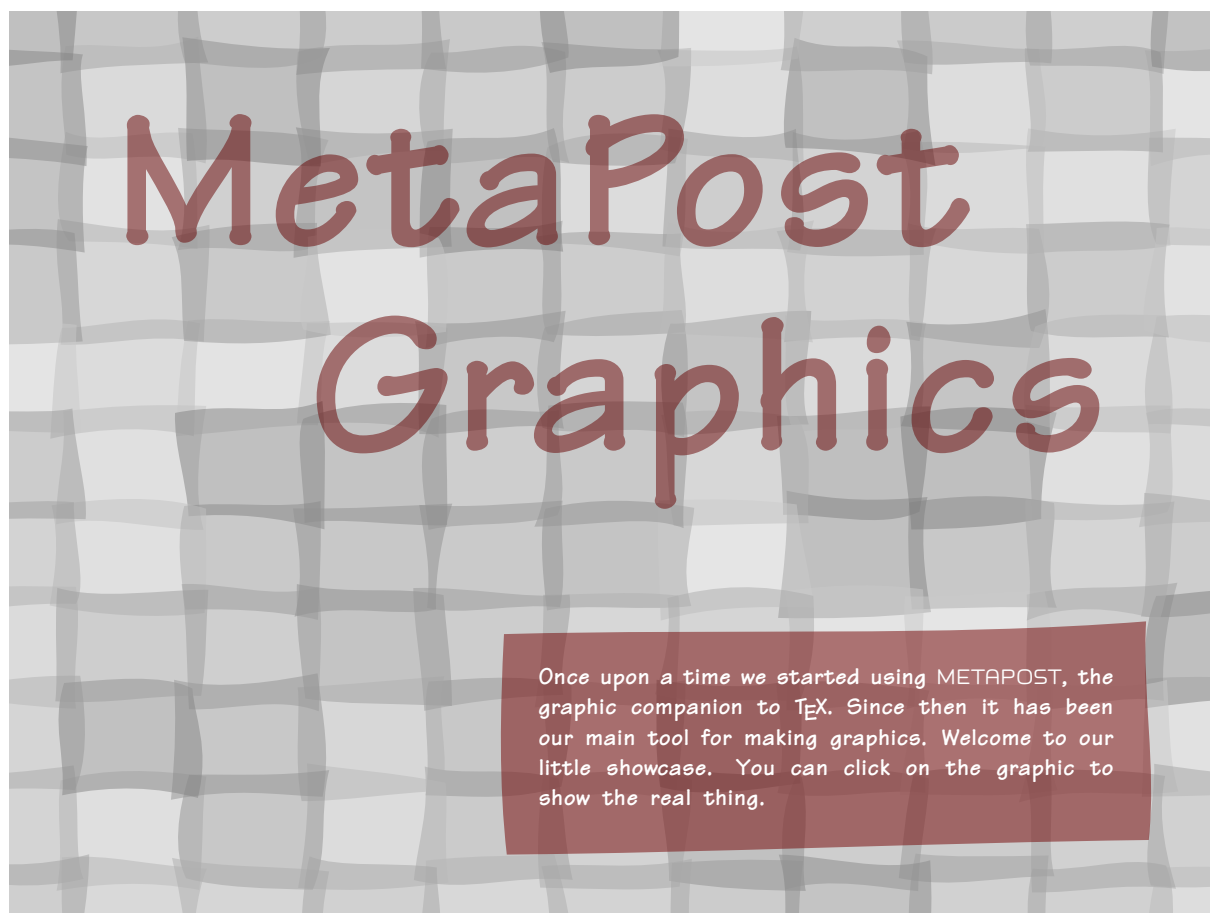


Figure II.IV http handling

We already had encountered the phenomena that when piping back results from LUA to T<sub>E</sub>X, we needed to take care of catcodes so that T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X we now have a convenient `\scantextokens` which has no side effects.

Once catcode tables were in place, and the relevant C<sub>ON</sub>T<sub>E</sub>X<sub>T</sub> code adapted, I could start playing with one of the trickier parts of T<sub>E</sub>X programming: typesetting T<sub>E</sub>X using T<sub>E</sub>X, or verbatim. Because in C<sub>ON</sub>T<sub>E</sub>X<sub>T</sub> 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 T<sub>E</sub>X, 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 M<sub>K</sub>I<sub>V</sub> 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:

	$\text{\TeX}$ normal	$\text{\TeX}$ 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  $\text{\TeX}$  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  $\text{\TeX}$  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  $\text{\CONTeXt}$  XML parser is highly optimized. However, if we want to load a whole XML file, for instance the formal  $\text{\CONTeXt}$  interface specification `cont-en.xml`, then we can bring down loading time (as well as  $\text{\TeX}$  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  $\text{\LUA\TeX}$ , 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  $\text{\CONTeXt}$  a bit will know that it may need multiple passes to typeset a document.  $\text{\CONTeXt}$  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 ( $\text{\CONTeXt}$  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 L<sup>A</sup>T<sub>E</sub>X but 8.7 seconds with traditional P<sup>D</sup>F<sub>E</sub>X. 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 millions of positions.



### III Initialization revised

Initializing L<sup>A</sup>T<sub>E</sub>X 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 originals 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 K<sub>P</sub>S<sub>E</sub> 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 T<sub>E</sub>X trees. Among the reasons why we want to replace K<sub>P</sub>S<sub>E</sub> are the facts that we want to access ZIP files, do more specific file searches, use HTTP, FTP and whatever comes around, integrate C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T 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 L<sup>A</sup>T<sub>E</sub>X code, we're talking about 0.3 seconds for P<sub>D</sub>F<sub>T</sub>E<sub>X</sub>. One would expect a L<sup>A</sup>UA 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 K<sub>P</sub>S<sub>E</sub>, 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 L<sup>A</sup>T<sub>E</sub>X. The method described here is developed for C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T but is not limited to this macro package; when one tells T<sub>E</sub>X<sub>E</sub>X<sub>E</sub>C to generate formats using the **--luatex** directive, it will generate the C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T formats as well as M<sub>P</sub>T<sub>O</sub>P<sub>D</sub>EF using this engine.

For practical reasons, the Lua based IO handler is K<sub>P</sub>S<sub>E</sub> compliant. This means that the normal **texmf.cnf** and **ls-R** files can be used. However, their content is converted in a more L<sup>A</sup>UA friendly way. Although this can be done at runtime, it makes more sense to do this in advance using L<sup>A</sup>U<sub>T</sub>O<sub>O</sub>L<sub>S</sub>. The files involved are:

input	raw input	runtime input	runtime fallback
	<b>ls-R</b>	<b>files.luc</b>	<b>files.lua</b>
<b>texmf.lua</b>	<b>temxf.cnf</b>	<b>configuration.luc</b>	<b>configuration.lua</b>

In due time L<sup>A</sup>U<sub>T</sub>O<sub>O</sub>L<sub>S</sub> 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 L<sup>A</sup>UA 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 L<sup>A</sup>T<sub>E</sub>X. For C<sup>O</sup>N<sup>T</sup>E<sup>X</sup>T this is normally done with the T<sub>E</sub>X<sub>EXEC</sub> wrapper, for instance:

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

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

- L<sup>A</sup>T<sub>E</sub>X 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, C<sup>O</sup>N<sup>T</sup>E<sup>X</sup>T will preload all LUA code and store them in the bytecode table provided by L<sup>A</sup>T<sub>E</sub>X. 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 T<sub>E</sub>X<sub>EXEC</sub> 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 T<sub>E</sub>X. In due time we will provide a few files that will making work with LUA more convenient in Plain T<sub>E</sub>X, but at this moment you can already use for instance `\directlua`.

In case you wonder how this is related to CON<sub>T</sub>E<sub>X</sub>T, well only to the extend that it uses a couple of rather generic CON<sub>T</sub>E<sub>X</sub>T related LUA files.

CON<sub>T</sub>E<sub>X</sub>T users can best use T<sub>E</sub>X<sub>EXEC</sub> 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 CON<sub>T</sub>E<sub>X</sub>T. Keep in mind that LUATOOLS does not set up user specific configurations, for instance the `--all` switch in T<sub>E</sub>X<sub>E</sub>EXEC will set up all patterns.

I know that it sounds a bit messy, but till we have a more clear picture of where L<sub>U</sub>A<sub>T</sub>E<sub>X</sub> is heading this is the way to proceed. The average CON<sub>T</sub>E<sub>X</sub>T user won't notice those details, because T<sub>E</sub>X<sub>E</sub>EXEC 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 T<sub>E</sub>X'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 took place right after the form was received by the web server.

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

By combining LUA with T<sub>E</sub>X, we can do the conversion from calculator math to T<sub>E</sub>X immediately, without auxiliary programs or complex parsing using T<sub>E</sub>X macros.

## tex

In a `CONTEX`T 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 `CONTEX`T 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 `CONTEX`T 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 `(1|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

LUA<sub>T</sub><sub>E</sub><sub>X</sub> 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 T<sub>E</sub>X 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, LUA<sub>T</sub><sub>E</sub><sub>X</sub> 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 LUA<sub>T</sub><sub>E</sub><sub>X</sub> code as well as the CON<sub>T</sub><sub>E</sub><sub>X</sub><sub>T</sub> 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. LUA<sub>T</sub><sub>E</sub><sub>X</sub> 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 T<sub>E</sub>X was doomed to be complex, and the whole idea behind L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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 T<sub>E</sub>X macro packages have always done that.

Yet another (more obscure) kind of remapping concerns those special T<sub>E</sub>X characters. If we use a traditional T<sub>E</sub>X 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  $\text{\TeX}$  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  $\text{\CONTeXt}$ , for performance reasons we load such files into memory, which means that we also need to provide a special reader to  $\text{\TeX}$ . 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  $\text{\CONTeXt}$ . Keep in mind that we overload the standard input methods and therefore have complete control over how  $\text{\LUA\TeX}$  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  $\text{\LUA\TeX}$  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  $\text{\CONTeXt MkIV}$  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  $\text{\LUA\TeX}$  moved on to UTF input does not mean that all the tools that users use and the files that they have produced over decades automatically 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  $\text{\TeX}$  we can do as before, but when we let  $\text{\LUA}$  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 T<sub>E</sub>X 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 OPEN<sub>TYPE</sub> fonts are the fashion, we still need to deal with T<sub>E</sub>X's native font machinery. Although Latin Modern and the T<sub>E</sub>X Gyre collection will bring us many free OPEN<sub>TYPE</sub> fonts, we can be sure that for a long time T<sub>Y</sub>P<sub>E</sub>T variants will be used as well, and when one has lots of bought fonts, replacing them with OPEN<sub>TYPE</sub> updates is not always an option. And so, reimplementing the readers for T<sub>E</sub>X 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 CON<sub>T</sub>E<sub>X</sub>T M<sub>K</sub>I<sub>V</sub>) 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,  $\text{LUA}\TeX$  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  $\text{LUA}$  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  $\text{DVI}$  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  $\text{TFM}$  table--- I started playing with constructing  $\text{TFM}$  tables directly. Because  $\text{CON}\TeX$  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  $\text{TFM}$  files out of our tree and construct the tables using the  $\text{AFM}$  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  $\text{LUA}\TeX$  into using  $\text{AFM}$  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  $\text{TFM}$  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 T<sub>E</sub>X 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 T<sub>E</sub>Xie 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 T<sub>E</sub>X'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 T<sub>E</sub>X 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 CON<sub>T</sub>E<sub>X</sub>T 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 T<sub>E</sub>X trees pretty well. Also, now that we have font related tables, we can start moving tables built out of T<sub>E</sub>X 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 an 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 L<sup>A</sup>T<sub>E</sub>X how to prepare and embed a file).

Eventually L<sup>A</sup>T<sub>E</sub>X 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 C<sup>O</sup>N<sup>T</sup>E<sup>X</sup>T 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 C<sup>O</sup>N<sup>T</sup>E<sup>X</sup>T 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 L<sup>A</sup>T<sub>E</sub>X . . . they will go away anyway.

## OTF & TTF

One of the reasons for starting the L<sup>A</sup>T<sub>E</sub>X development was that we wanted to be able to use O<sup>P</sup>E<sup>N</sup>T<sup>Y</sup>P<sup>E</sup> (and T<sup>R</sup>U<sup>E</sup>T<sup>Y</sup>P<sup>E</sup>) fonts in P<sup>D</sup>F<sup>T</sup>E<sup>X</sup>. As a prelude (and kind of transition) we first dealt with T<sup>Y</sup>P<sup>E</sup>1 using either T<sup>F</sup>M or A<sup>F</sup>M. For T<sub>E</sub>X 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 T<sub>E</sub>X kernel, but think of HZ or protruding as provided by P<sup>D</sup>F<sup>T</sup>E<sup>X</sup>: 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.

O<sup>P</sup>E<sup>N</sup>T<sup>Y</sup>P<sup>E</sup> 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 T<sub>E</sub>Xies know, other relationships exist as well, for instance between content and visualization.

Therefore, it will be no surprise that L<sup>A</sup>T<sub>E</sub>X does not simply implement the O<sup>P</sup>E<sup>N</sup>T<sup>E</sup><sub>Y</sub><sup>E</sup> 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 fulfil the demands of such fonts and more. The glue between both is done with L<sup>U</sup>A. 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 O<sup>T</sup><sub>T</sub>F or T<sub>T</sub>F font. From this table, a font table is constructed that is passed on to T<sub>E</sub>X. 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 L<sup>A</sup>T<sub>E</sub>X system, O<sup>P</sup>E<sup>N</sup>T<sup>E</sup><sub>Y</sub><sup>E</sup> 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 T<sub>E</sub>X 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 L<sup>U</sup>A tables in a well organized way.

Given that users expect O<sup>P</sup>E<sup>N</sup>T<sup>E</sup><sub>Y</sub><sup>E</sup> features to be supported, how do we provide an interface. In C<sup>O</sup>N<sup>T</sup>E<sup>X</sup><sub>T</sub> 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 explicitly and a new custom introduced by X<sub>Y</sub>T<sub>E</sub>X to enhance fontname with directives. Traditional T<sub>E</sub>X provides:

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

X<sub>Y</sub>T<sub>E</sub>X 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. L<sup>A</sup>T<sub>E</sub>X 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 X<sub>Y</sub>TeX 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 L<sup>A</sup>TeX has the backend built in, it is capable of handling virtual fonts information.

In L<sup>A</sup>TeX 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 (in real 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 prize and In the experimental CON<sub>T</sub>E<sub>X</sub>T code we used calls like where **demo** is an installed feature.

Hi there, this is the first (number 1) example of playing with Virtual Fonts, some neat feature of T<sub>E</sub>X, 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 T<sub>E</sub>X's internals.

While developing this functionality and especially when Taco was programming the back-end functionality, we used more sane M<sub>K</sub>IV code. Think of (still L<sub>U</sub>A) definitions like:

```
\ctxlua {
  fonts.define.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} \test
\acute \rcaron \adoublegrave \char65
```

This shows up as:

éřäB

Here the @ tells the (new) CON<sub>T</sub>E<sub>X</sub>T font handler what constructor should be used.

Because some testers already have X<sub>Y</sub>T<sub>E</sub>X font support files, we also support a X<sub>Y</sub>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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/T<sub>E</sub>X files or O<sub>P</sub>E<sub>N</sub>T<sub>E</sub>X fonts will be used when available.

A couple of years from now, we may throw away some L<sup>A</sup>U<sub>A</sub> code related to encodings.

## files

T<sub>E</sub>X distributions tend to be rather large, both in terms of files and bytes. Fonts take most of the space. The merged T<sub>E</sub>XLive 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 C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T distribution spans 1200 files and 20 MBytes and a bit more when third party modules are taken into account. The fonts in T<sub>E</sub>XLive are distributed as follows:

format	files	bytes
AFM	1.769	123.068.970
TFM	10.613	44.915.448
VF	3.798	6.322.343
T <sub>E</sub> X <sub>1</sub>	2.904	180.567.337
TRUETYPE	22	1.494.943
O <sub>P</sub> E <sub>N</sub> T <sub>E</sub> X	144	17.571.732
ENC	268	782.680
MAP	406	6.098.982
O <sub>F</sub> M	39	10.309.792
O <sub>V</sub> F	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 cached (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 virtual 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.tfm.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).

c   c̣   C̣   C̣   ˇs   ´e   ¨a   ¨u   ˇO   ˇI   .b  
*c   c̣   C̣   C̣   ˇs   ´e   ¨a   ¨u   ˇO   ˇI   .b*

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

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

Without the fallbacks we get:

c   c̣   C̣   C̣   š   é   ä   ü  
*c   c̣   C̣   C̣   š   é   ä   ü*

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

c   c̣   C̣   C̣   š   é   ä   ü   ˇO   ˇI   .b  
*c   c̣   C̣   C̣   š   é   ä   ü   ˇO   ˇI   .b*

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 T<sub>E</sub>X has been designed for speed and a small memory footprint. Today's 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 L<sup>A</sup>T<sub>E</sub>X is not only suitable for wide fonts, but also does away with some of the optimizations in the T<sub>E</sub>X code that complicate extensions, it has a larger footprint than P<sub>D</sub>F<sub>E</sub>T<sub>E</sub>X. When implementing the O<sub>P</sub>E<sub>N</sub>T<sub>Y</sub>P<sub>E</sub> font basics, we did quite some tests with respect to memory usage. Getting the numbers right is non trivial because the L<sup>A</sup> garbage collector is interfering. For instance, on my machine a test file with the regular C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T setup of Latin Modern fonts made L<sup>A</sup> allocate 130 MB, while the same run on Taco's machine took 100 MB.

When a font data table is constructed, it is handed over to T<sub>E</sub>X, and turned into the internal font data structures. During the construction of that T<sub>A</sub>B<sub>L</sub>E at the L<sup>A</sup> end, C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T M<sub>K</sub>I<sub>V</sub> 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 (O<sub>P</sub>E<sub>N</sub>T<sub>Y</sub>P<sub>E</sub>) takes around 800 KB. However, more interesting was that by adding this snippet of testcode which duplicated 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 T<sub>E</sub>X 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 L<sup>A</sup>, it no longer makes sense to let T<sub>E</sub>X buffer fonts. In plain T<sub>E</sub>X one finds these magic

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

lines. The second definition obscures the first, but the **cmr10** stays loaded.

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

These two definitions make T<sub>E</sub>X load the font only once. However, since we can now delegate loading to L<sub>U</sub>A, T<sub>E</sub>X no longer helps us there. For instance, T<sub>E</sub>X has no knowledge to what extent this **cmr10** font has been manipulated and therefore both instances may actually differ.

When you use a callback to define the font, T<sub>E</sub>X passes a font id number. You can use this number as a reference to a loaded font (that is, passed to T<sub>E</sub>X). If instead of a table, you return a number, T<sub>E</sub>X will reuse the already loaded font. This feature can save you a lot of time, especially when a macro package (like CON<sub>T</sub>E<sub>X</sub>T) 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 L<sub>U</sub>A end, but there one needs to take into account more than just the scaled instance. Think of O<sub>P</sub>E<sub>N</sub>T<sub>Y</sub>P<sub>E</sub> 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 page is 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 T<sub>E</sub>X 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  $\text{\TeX}$  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 truck-load of other codes, with lists made out of boxes, skips, kerns and penalties but even they cannot look much deeper into  $\text{\TeX}$ 's internals. Their deeper understanding comes from reading the  $\text{\TeX}$ book or even looking at the source code.

When someone enters the magic world of  $\text{\TeX}$  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  $\text{\TeX}$  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  $\text{\LaTeX}$  shed light on this.

In a moment we will show examples of how  $\text{\LaTeX}$  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  $\text{\TeX}$  sees your input.

Hi there!

Hi there!

cmd	chr	id	name
-----	-----	----	------

```

letter      72  H
letter     105  i
spacer      32
letter     116  t
letter     104  h
letter     101  e
letter     114  r
letter     101  e
other_char  33  !

```

Here we see three kind of 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**

Hans & Taco use LuaTeX!

cmd	chr	id	name
letter	72	H	
letter	97	a	
letter	110	n	
letter	115	s	
spacer	32		
char_given	38	131112	&
spacer	32		
letter	84	T	
letter	97	a	
letter	99	c	
letter	111	o	
spacer	32		
letter	117	u	
letter	115	s	
letter	101	e	
spacer	32		
letter	76	L	
letter	117	u	
letter	97	a	
long_call	470538	131700	TeX
char_num	0	132590	char
other_char	51	3	
other_char	51	3	
relax	1114112	134452	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 T<sub>E</sub>X 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
```

**Hans**

Taco

cmd	chr	id	name
start_par	0	158918	noindent
left_brace	123		
long_call	368999	131372	bf
letter	72	H	
letter	97	a	
letter	110	n	
letter	115	s	
right_brace	125		
spacer	32		
par_end	1114112	131830	par
make_box	123	132640	hbox
left_brace	123		
letter	84	T	
letter	97	a	
letter	99	c	
letter	111	o	
right_brace	125		
spacer	32		
par_end	1114112	144234	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
```

before after 10.0pt

cmd	chr	id	name
letter	98	b	
letter	101	e	
letter	102	f	
letter	111	o	

letter	114	r	
letter	101	e	
spacer	32		
assign_box_dir	2	134262	dimen
other_char	50	2	
other_char	61	=	
other_char	49	1	
other_char	48	0	
letter	112	p	
letter	116	t	
spacer	32		
letter	97	a	
letter	102	f	
letter	116	t	
letter	101	e	
letter	114	r	
spacer	32		
top_bot_mark	0	131847	the
assign_box_dir	2	134262	dimen
other_char	50	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

before 

whatever
----------

 after

cmd	chr	id	name
letter	98	b	
letter	101	e	
letter	102	f	
letter	111	o	
letter	114	r	
letter	101	e	
spacer	32		
long_call	187371	1326018	inframed
other_char	91	[	
letter	119	w	
letter	105	i	
letter	100	d	
letter	116	t	
letter	104	h	

other_char	61	=
other_char	51	3
letter	99	c
letter	109	m
other_char	93	]
left_brace	123	
letter	119	w
letter	104	h
letter	97	a
letter	116	t
letter	101	e
letter	118	v
letter	101	e
letter	114	r
right_brace	125	
spacer	32	
letter	97	a
letter	102	f
letter	116	t
letter	101	e
letter	114	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**

compound-word

cmd	chr	id	name
letter	99	c	
letter	111	o	
letter	109	m	
letter	112	p	
letter	111	o	
letter	117	u	
letter	110	n	
letter	100	d	
long_call	135958	125	
other_char	45	-	
long_call	135958	125	
letter	119	w	

letter	111	o
letter	114	r
letter	100	d

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

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

hm, Hello World!

cmd	chr	id	name
letter	104	h	
letter	109	m	
other_char	44	,	
spacer	32		
the	23	183917	directlua
other_char	48	0	
spacer	32		
left_brace	123		
spacer	32		
letter	116	t	
letter	101	e	
letter	120	x	
other_char	46	.	
letter	115	s	
letter	112	p	
letter	114	r	
letter	105	i	
letter	110	n	
letter	116	t	
other_char	40	(	
other_char	34	"	
letter	72	H	
letter	101	e	
letter	108	l	
letter	108	l	
letter	111	o	
spacer	32		
letter	87	W	
letter	111	o	
letter	114	r	
letter	108	l	
letter	100	d	

```

other_char    33  !
other_char    34  "
other_char    41  )
spacer        32
right_brace   125

```

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.

```
[B][A]
```

cmd	chr	id	name
set_box	0	131778	def
expand_after	0	132496	Test
mac_param	35		
other_char	49	1	
mac_param	35		
other_char	50	2	
left_brace	123		
other_char	91	[	
mac_param	35		
other_char	50	2	
other_char	93	]	
other_char	91	[	
mac_param	35		
other_char	49	1	
other_char	93	]	
right_brace	125		
spacer	32		
expand_after	0	132496	Test
left_brace	123		
letter	65	A	
right_brace	125		
left_brace	123		
letter	66	B	
right_brace	125		

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. A stupid callback looks like:

```
callback.register('token_filter', token.get_next)
```

In principle you can call `token.get_next` anytime you want to intercept a token. In that case you can feed back tokens into T<sub>E</sub>X by using a trick like:

```
function tex.printlist(data)
  callback.register('token_filter', function ()
    callback.register('token_filter', nil)
    return data
  end)
end
```

Another example of usage is:

```
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)
```

There is a whole repertoire of related functions, one is `token.create`, which can be used as:

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

This results in: ?

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

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

Unnesting is needed because the result of the `letters` call is a table, and the `print-list` function wants a flattened table.

The result looks like: 12345



cmd	chr	id	name
make_box	123	132640	hbox
left_brace	123		
letter	49	1	
letter	50	2	
letter	51	3	
letter	52	4	
letter	53	5	
right_brace	125		

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.

`\hbox{12345\hbox{54321}}`

can also be done by saying:

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

or under CON<sub>T</sub>EX<sub>T</sub>'s basic catcode regime:

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

If you like it the hard way:

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

This method may attract those who dislike the traditional T<sub>E</sub>X syntax for doing the same thing. Okay, a carefull reader will notice that reversing the string in T<sub>E</sub>X takes a bit more trickery, so ...



## 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  $\text{\TeX}$ –LUA pair was up to the task. And so we were quite confident that processing nodes was not only an important aspect of  $\text{\texttt{LUA}\TeX}$  but also quite feasible in terms of performance (after all we needed it in order to deal with advanced typesetting of Arab). When Taco was dealing with the  $\text{\TeX}$  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  $\text{\texttt{METAPOST}}$  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.

---

<code>gsub</code>	<code>lpeg</code>
-------------------	-------------------

---

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 T<sub>E</sub>X (PDF) code in a table and passed it in its concatenated form to T<sub>E</sub>X. Because the LUA to T<sub>E</sub>X interface is by now quite efficient we can just pass the intermediate results directly to T<sub>E</sub>X.

## file io

The repertoire 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 T<sub>E</sub>X 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 CON<sub>T</sub>E<sub>X</sub>T 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 T<sub>E</sub>X sees them, in which case T<sub>E</sub>X 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 L<sub>U</sub>A<sub>T</sub><sub>E</sub>X 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 sure that when half a second runtime is spent on reading a file, processing may take minutes. If one wants to squeeze more performance out of this part, it's always an option to write special libraries for that, but this is beyond standard L<sup>A</sup>T<sub>E</sub>X. We found out that the speed of loading data from files in L<sup>A</sup>UA is mostly related to the small size of L<sup>A</sup>UA'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 M<sub>K</sub>IV we allocate quite some memory due to font handling, we were pretty sure that handling T<sub>E</sub>X'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 is 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 the new `lpeg` based METAPost converter, we ran into problems. For table intensive operations, temporarily 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 disastrous. Because no other LUA calls happen between these calls, the garbage collector is never invoked at all. As a result, memory grew 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 L<sup>A</sup>T<sub>E</sub>X 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	chr	id	name
set_box	0	131778	def
expand_after	0	1335790	SwapChars
mac_param	35		
other_char	49	1	
mac_param	35		
other_char	50	2	
left_brace	123		
mac_param	35		
other_char	50	2	
mac_param	35		
other_char	49	1	
right_brace	125		

## IX Nodes and attributes

### introduction

Here we will tell a bit about the development of node access in L<sup>A</sup>T<sub>E</sub>X. We will also introduce attributes, a feature closely related to nodes. We assume that you are somewhat familiar with T<sub>E</sub>X'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 T<sub>E</sub>X 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, and 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 T<sub>E</sub>X 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 T<sub>E</sub>X end of the game.

Side note concerning speed: when memory of processing speed is low, runtime can increase five to tenfold compared to PDF<sub>T</sub><sub>E</sub>X when one does intensive node manipulations. This is due to garbage collection at the LUA end and memory (de)allocation at the T<sub>E</sub>X end. There is not much we can do about that. Interfacing has a price and hardware is more powerful than when T<sub>E</sub>X was written. Processing the T<sub>E</sub>X book using no callbacks is not that much slower than using a traditional T<sub>E</sub>X 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 T<sub>E</sub>X that it can keep the original list. When the list is empty, the callback function can return the value `false`. This signals T<sub>E</sub>X 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 to 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 T<sub>E</sub>X 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 T<sub>E</sub>X 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 T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X, 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 T<sub>E</sub>X 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 than messing around with user-data 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 tn timer = type(nv)
        if tn == "string" or tn == "number" then
          tt[v] = nv
        else -- userdata
          tt[v] = nodes.totable(nv)
        end
      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 T<sub>E</sub>X 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 T<sub>E</sub>X removes these kerns and inserts them later (but then based on the information stored in the font).

The reason why T<sub>E</sub>X 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`, T<sub>E</sub>X 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 T<sub>E</sub>X 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 T<sub>E</sub>X 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 T<sub>E</sub>X is determining feasible break-points (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`.

```
\attributedef\aa=\numexpr\attdefcounter+2\relax % no clash
\attributedef\ab=\numexpr\attdefcounter+3\relax
\visualizenextnodes \hbox {\aa1 T{\ab3\aa2 E}X}
```

For the sake of this example, we start the allocation at 200 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
```

```

t={
    type="hlist",
    attr={
        type="attribute_list",
        id=44,
        next={
            type="attribute",
            id=42,
            next={
                type="attribute",
                id=42,
                number=6,
                value=1,
                next={
                    type="attribute",
                    id=42,
                    number=7,
                    value=3,
                },
            },
        },
    },
    width=1135419,
    height=440470,
    list={
        type="glyph",
        id=33,
        attr={
            type="attribute_list",
            id=44,
            next={
                type="attribute",
                id=42,
                next={
                    type="attribute",
                    id=42,
                    number=6,
                    value=1,
                    next={
                        type="attribute",
                        id=42,
                        number=7,
                        value=3,
                    },
                },
            },
        },
        next={
            type="attribute",
            id=42,
            next={
                type="attribute",
                id=42,
                number=25,
                value=2,
                next={
                    type="attribute",
                    id=42,
                    number=26,
                    value=3,
                },
            },
        },
    },
    value=1,
    },
    },
    char=84,
    font=81,
    lang=2,
    left=2,
    right=3,
    uchyph=1,
    next={
        type="glyph",
        id=33,
        attr={
            type="attribute_list",
            id=44,
            next={
                type="attribute",
                id=42,
                next={
                    type="attribute",
                    id=42,
                    number=6,
                    value=1,
                    next={
                        type="attribute",
                        id=42,
                        number=7,
                        value=3,
                    },
                },
            },
        },
        next={
            type="attribute",
            id=42,
            next={
                type="attribute",
                id=42,
                number=25,
                value=1,
            },
        },
    },
    char=69,
    font=81,
    lang=2,
    left=2,
    right=3,
    uchyph=1,
    next={
        type="glyph",
        id=33,
        attr={
            type="attribute_list",
            id=44,
            next={
                type="attribute",
                id=42,
                next={
                    type="attribute",
                    id=42,
                    number=7,
                    value=3,
                },
            },
        },
        next={
            type="attribute",
            id=42,
            number=25,
            value=1,
        },
    },
    char=88,
    font=81,
    lang=2,
    left=2,
    right=3,
    uchyph=1,
    },
    },
    },
}

```

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

The previous function can be used in tests like:

```
local total = 0
while n do
    if has_attribute(n,200) then
        total = total + 1
    end
    n = n.next
end
texio.write_nl(string.format("attribute 200 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 L<sup>A</sup>T<sub>E</sub>X 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 T<sub>E</sub>X 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 `CONTEX`T, attributes will find their way rather soon in `CONTEX`T 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 T<sub>E</sub>Xbook, 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 T<sub>E</sub>X in order to master this kind of trickyness.

In this chapter I will show a few dirty L<sup>A</sup>T<sub>E</sub>X 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 T<sub>E</sub>X 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 T<sub>E</sub>X 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 T<sub>E</sub>X 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.wd[0]
  tex.sprint("\string\\the\string\\dimen0")
}
```

This gives: 0.opt. 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.wd[0])
  end
end
```

```

        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 `thearafterlua`. 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))
    theafterlua(tex.wd[0],tex.ht[0],tex.dp[0])
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:

```

do

    delayed = { } -- could also be done with closures

```

```

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

function lua.flush_delayed(...)
    local t = delayed
    delayed = { }
    for _, fun in ipairs(t) do
        fun(...)
    end
end

function lua.flush(...)
    tex.sprint("\directlua 0 {lua.flush_delayed(" ..
        table.concat({...},',' .. ")}")
end

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))
    lua.flush(tex.wd[0],tex.ht[0],tex.dp[0])
end

```





# XI Going beta

## introduction

We're closing in on the day that we will go beta with L<sup>A</sup>T<sub>E</sub>X (end of July 2007). By now we have a rather good picture of its potential and to what extent L<sup>A</sup>T<sub>E</sub>X will solve some of our persistent problems. Let's first summarize our reasons for and objectives with L<sup>A</sup>T<sub>E</sub>X.

- The world has moved from 8 bits to 32 bits and more, and this is quite noticeable in the arena of fonts. Although T<sub>E</sub>X fonts could host more than 256 glyphs, the associated technology was limited to 256. The advent of O<sub>P</sub>E<sub>N</sub>T<sub>E</sub>X 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 U<sub>N</sub>I<sub>C</sub>O<sub>D</sub>E is replacing 8 bit based encoding vectors and code pages (input regimes). The most popular and rather efficient U<sub>T</sub>F<sub>8</sub> encoding has become a de factor standard in document encoding and interchange.
- Although we can do real neat tricks with T<sub>E</sub>X, 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 L<sup>A</sup>T<sub>E</sub>X: a merge of the mainstream traditional T<sub>E</sub>X engines, stripped from broken or incomplete features and opened up to an embedded L<sup>A</sup>UA scripting engine.

We will describe the impact of this new engine by starting from its core components reflected in the specific L<sup>A</sup>UA interface libraries. Missing here is embedded support for M<sub>E</sub>-T<sub>A</sub>P<sub>O</sub>ST, because it's not yet there (apart from the fact that we use L<sup>A</sup>UA to convert M<sub>E</sub>T<sub>A</sub>P<sub>O</sub>ST graphics into T<sub>E</sub>X). Also missing is the interfacing to the P<sub>D</sub>F 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 C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T as testbed, we will refer to the L<sup>A</sup>T<sub>E</sub>X aware version of this macro package, M<sub>K</sub>I<sub>V</sub>, but most conclusions are rather generic.

## tex internals

In order to manipulate T<sub>E</sub>X'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 need 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 T<sub>E</sub>X 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 T<sub>E</sub>X machinery.

What will be the impact of access on CON<sub>T</sub>EXT MkIV? This is yet hard to tell. In a later stage of the development, when parts of the T<sub>E</sub>X 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 T<sub>E</sub>X'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 LUAT<sub>E</sub>X: here both worlds communicate with each other. A callback is a place in the T<sub>E</sub>X 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 `CONTEX`T 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, `CONTEX`T 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 `CONTEX`T, 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 `CONTEX`T 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 than `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 CON<sub>T</sub>EX<sub>T</sub> 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 T<sub>E</sub>X's error handling.

Another consequence of this will be that we will be able to package T<sub>E</sub>X 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 OPEN<sub>T</sub>YPE involves more than just loading yet another font format. Of course loading an OPEN<sub>T</sub>YPE 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 or combining sequences into other sequences and finally resolving them to one or more glyphs.

Given the numerous options we will have to spend quite some time on extending CON<sub>T</sub>EX<sub>T</sub> with new features. Instead of defining more and more font instances (the traditional T<sub>E</sub>X way of doing things) we will provide 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 already in place (ligature building and such). Here we also have to decide what we let T<sub>E</sub>X do and what we do by processing node lists. For instance kerning and ligature building can either be done by T<sub>E</sub>X or by L<sub>U</sub>A. Given the fact that T<sub>E</sub>X does some juggling with character kerning while determining hyphenation points, we can as well disable T<sub>E</sub>X's kerning and let L<sub>U</sub>A handle it. Thereby T<sub>E</sub>X only has to deal with paragraph building. (After all, we need to leave T<sub>E</sub>X 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 CON<sub>T</sub>EX<sub>T</sub> 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 T<sub>E</sub>X. This dependency will go away. In MkII we already ship UTF8 based patterns for 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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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 a price 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 T<sub>E</sub>X 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 T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X, out of a sudden we can look into T<sub>E</sub>X's internals and manipulate them. Although I don't claim to be a real expert on these internals, even after over a decade of T<sub>E</sub>X 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 T<sub>E</sub>X is returning. It all has to do with the way T<sub>E</sub>X collects information and when it decides to act. In regular T<sub>E</sub>X 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 `CONTEX` 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 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 `CONTEX` support both good old `TEX` (represented by `PDFTEX`), `XYTEX` (a `UNICODE` and `OPENTEX` 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 `CONTEX` 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  $\text{CON}\mathcal{T}\mathcal{E}\mathcal{X}\mathcal{T}$  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  $\text{CON}\mathcal{T}\mathcal{E}\mathcal{X}\mathcal{T}$ . 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  $\text{CON}\mathcal{T}\mathcal{E}\mathcal{X}\mathcal{T}$  column mechanism is grid aware, we may as well add that and/or implement an additional mechanism.

A side effect of being able to do this in  $\text{LUA}\mathcal{T}\mathcal{E}\mathcal{X}$  is that the code taken from  $\text{PDF}\mathcal{T}\mathcal{E}\mathcal{X}$  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 issues, 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- $\mathcal{T}\mathcal{E}\mathcal{X}$  code will be generic in nature. Eventually  $\text{CON}\mathcal{T}\mathcal{E}\mathcal{X}\mathcal{T}$  will become a top layer on what we internally call  $\text{META}\mathcal{T}\mathcal{E}\mathcal{X}$ , a collection of kernel modules that one can use to build specialized macro packages. To some extent  $\text{META}\mathcal{T}\mathcal{E}\mathcal{X}$  can be for  $\text{LUA}\mathcal{T}\mathcal{E}\mathcal{X}$  what plain is for  $\mathcal{T}\mathcal{E}\mathcal{X}$ . But if and how fast this will be reality depends on the amount of time that we (and other members of the  $\text{CON}\mathcal{T}\mathcal{E}\mathcal{X}\mathcal{T}$  development team) can allocate to this.





## XII Zapfing fonts

### 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 replaced 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 complicates 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 CON<sub>T</sub>E<sub>X</sub>T users normally use when testing layouts or new features, a quote from E.R. Tufte and one from Hermann Zapf. The T<sub>E</sub>X code shows how features are set in CON<sub>T</sub>E<sub>X</sub>T.

```
\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, synopsise, 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 a special glyph representing  $\%$
- in the input stream this is the character sequence `c/o`
- so there must 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 L<sup>A</sup>T<sub>E</sub>X 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 pity 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:

<code>three fraction four.2</code>	<code>threequarters</code>	$\frac{3}{4}$
<code>three fraction four</code>	<code>threequarters</code>	$\frac{3}{4}$
<code>d r</code>	<code>d_r</code>	$dr$
<code>e period</code>	<code>e_period</code>	$e\cdot$
<code>f i</code>	<code>fi</code>	$fi$
<code>f l</code>	<code>fl</code>	$fl$
<code>f f i</code>	<code>f_f_i</code>	$ffi$
<code>f t</code>	<code>f_t</code>	$ft$

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 `CONTEX` 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 applied 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 L<sup>A</sup>T<sub>E</sub>X 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_1_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:

<b>none</b>	<i>on the synthesis</i>	<i>winnow the wheat</i>
<b>calt</b>	<i>on the synthesis</i>	<i>winnow the wheat</i>
<b>clig</b>	<i>on the synthesis</i>	<i>winnow the wheat</i>
<b>dlig</b>	<i>on the synthesis</i>	<i>winnow the wheat</i>
<b>liga</b>	<i>on the synthesis</i>	<i>winnow the wheat</i>

Here are Adam's recommendations with regards to the **dlig** feature: "The **dlig** feature is supposed to be used 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."

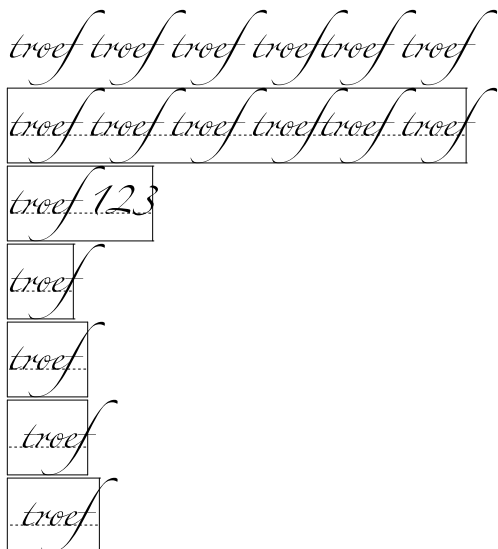
When testing features it helps to use words that look similar so next we will show some examples that were used. When we look at these examples, we need to understand that when a specific character representation is analyzed, the rules can take preceding and following characters into account. The rules 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 the rules is definitely non trivial.

When testing the implementation we ran into cases where the initial **t** showed up wrong, for instance in 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
```

*review review review, review*

itemize, review

*itemize, review,*

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

Welcome to Zappino

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.

synopsise versus synthesise 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:

<b>h</b>	<i>h</i>
<b>h.2</b>	<i>h</i>
<b>h.3</b>	<i>h</i>
<b>h.4</b>	<i>h</i>
<b>h.5</b>	<i>h</i>
<b>h.init</b>	<i>h</i>
<b>h.sups</b>	<i>h</i>
<b>h.sc</b>	<i>H</i>
<b>orn.73</b>	<i>h</i>

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 T<sub>E</sub>X 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 T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X.

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 T<sub>E</sub>X's default mechanism. When L<sup>A</sup>T<sub>E</sub>X and MkIV are beyond beta stage, we will use that mode by default.

<code>analyze=yes, language=dflt, mode=node, script=arab</code>	
---	---

الْحَمْدُ لِلَّهِ حَمْدًا مُعْتَرَفًا بِحَمْدِهِ، مُعْتَرَفًا مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ  
شَاكِرًا، وَلِحُسْنِ الْإِيْهِانِ أَشْرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوَةَ، وَالْخَيْرَ وَالشَّرَّ،  
وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

الْحَمْدُ لِلَّهِ حَمْدًا مُعْتَرَفًا بِحَمْدِهِ، مُعْتَرَفًا مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ  
شَاكِرًا، وَلِحُسْنِ الْإِيْهِانِ أَشْرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوَةَ، وَالْخَيْرَ وَالشَّرَّ،  
وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ، وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

Once these forms are identified, the `init`, `medi`, `fin` and `isol` features can be applied since they need this information. As you can see, different shapes show up. The vowels (marks in O<sup>P</sup>E<sup>N</sup>T<sup>E</sup>X speak) are not affected. It may not be entirely clear here, but these vowels don't have width.

```
analyze=yes, fina=yes, init=yes, isol=yes,  
language=dflt, medi=yes, mode=node, script=arab
```

لله

الْحَمْدُ لِلَّهِ حَمْدٌ مُعْتَرِفٌ بِحَمْدِهِ، مُعْتَرِفٌ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اللَّائِهِ  
نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوَةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ،  
وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

الْحَمْدُ لِلَّهِ حَمْدٌ مُعْتَرِفٌ بِحَمْدِهِ، مُعْتَرِفٌ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اللَّائِهِ  
نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوَةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ،  
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, fina=yes, init=yes, isol=yes,  
language=dflt, mark=yes, medi=yes, mode=node,  
script=arab
```

لله

الْحَمْدُ لِلَّهِ حَمْدٌ مُعْتَرِفٌ بِحَمْدِهِ، مُعْتَرِفٌ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اللَّائِهِ  
نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوَةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ،  
وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

الْحَمْدُ لِلَّهِ حَمْدٌ مُعْتَرِفٌ بِحَمْدِهِ، مُعْتَرِفٌ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اللَّائِهِ  
نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوَةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ،  
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, fina=yes, init=yes, isol=yes,  
language=dflt, mark=yes, medi=yes, mkmk=yes,  
mode=node, script=arab
```

لله

الْحَمْدُ لِلَّهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُعْتَرِفٍ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اللَّيْهِ  
 نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَاةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ،  
 وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

الْحَمْدُ لِلَّهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُعْتَرِفٍ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ اللَّيْهِ  
 نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَاةَ، 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, fina=yes, init=yes, isol=yes,  
kern=yes, language=dflt, mark=yes, medi=yes,  
mkmk=yes, mode=node, script=arab
```

لله

الْحَمْدُ لِلَّهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُعْتَرِفٍ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ  
 اللَّيْهِ نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَاةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَةَ،  
 وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

الْحَمْدُ لِلَّهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُعْتَرِفٍ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ  
 اللَّيْهِ نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَاةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ 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, fina=yes, init=yes,  
isol=yes, kern=yes, language=dflt, mark=yes,  
medi=yes, mkmk=yes, mode=node, script=arab
```

لله

الْحَمْدُ لِلَّهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُعْتَرِفٍ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ  
الِإِيَّهِ نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَاةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ،  
وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

الْحَمْدُ لِلَّهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُعْتَرِفٍ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ  
الِإِيَّهِ نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَاةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ،  
وَالْأَرْوَاحَ وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

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

```
analyze=yes, clig=yes, dlig=yes, fina=yes,  
init=yes, isol=yes, kern=yes, language=dflt,  
liga=yes, mark=yes, medi=yes, mkmk=yes,  
mode=node, rlig=yes, script=arab
```

الله

الْحَمْدُ لِلَّهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُعْتَرِفٍ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ الْإِيَّهِ  
نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَاةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ، وَالْأَرْوَاحَ  
وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

الْحَمْدُ لِلَّهِ حَمْدَ مُعْتَرِفٍ بِحَمْدِهِ، مُعْتَرِفٍ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ الْإِيَّهِ  
نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَاةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ، 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, clig=yes, curs=yes, dlig=yes,  
fina=yes, init=yes, isol=yes, kern=yes,  
language=dflt, liga=yes, mark=yes, medi=yes,  
mkmk=yes, mode=node, rlig=yes, script=arab
```

الله



الْحَمْدُ لِلّٰهِ حَمْدٌ مُّغْتَرِفٌ بِحَمْدِهِ، مُّغْتَرِفٌ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ الْإِيَّاهِ  
 نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوَةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ، وَالْأَزْوَاحَ  
 وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

الْحَمْدُ لِلّٰهِ حَمْدٌ مُّغْتَرِفٌ بِحَمْدِهِ، مُّغْتَرِفٌ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ الْإِيَّاهِ  
 نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوَةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ، وَالْأَزْوَاحَ  
 وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

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

```
analyze=yes, clig=yes, curs=yes, dlig=yes,
fina=yes, init=yes, isol=yes, kern=yes,
language=urd, liga=yes, mark=yes, medi=yes,
mkmk=yes, mode=node, rlig=yes, script=arab
```

لله

الْحَمْدُ لِلّٰهِ حَمْدٌ مُّغْتَرِفٌ بِحَمْدِهِ، مُّغْتَرِفٌ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ الْإِيَّاهِ  
 نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوَةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ، وَالْأَزْوَاحَ  
 وَالْأَجْسَامَ، وَالذِّكْرَ وَالنِّسْيَانَ.

الْحَمْدُ لِلّٰهِ حَمْدٌ مُّغْتَرِفٌ بِحَمْدِهِ، مُّغْتَرِفٌ مِنْ بَحَارِ مَجْدِهِ، بِلِسَانِ الثَّنَاءِ شَاكِرًا، وَلِحُسْنِ الْإِيَّاهِ  
 نَاشِرًا؛ الَّذِي خَلَقَ الْمَوْتَ وَالْحَيَوَةَ، وَالْخَيْرَ وَالشَّرَّ، وَالنَّفْعَ وَالضَّرَّ، وَالسُّكُونَ وَالْحَرَكَهَ، وَالْأَزْوَاحَ  
 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 a 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 `CONTEX`T wiki.

## XIV Colors redone

### introduction

Color support has been present in `CONTEX` right from the start and support has been gradually 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
	red		r=1.000 g=0.000 b=0.000
	green		r=0.000 g=1.000 b=0.000
	blue		r=0.000 g=0.000 b=1.000
	yellow		c=0.000 m=0.000 y=1.000 k=0.000
	magenta		c=0.000 m=1.000 y=0.000 k=0.000
	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}
\freezeclorstrue
\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]
```



Rasters are still supported but normally one will use colors:

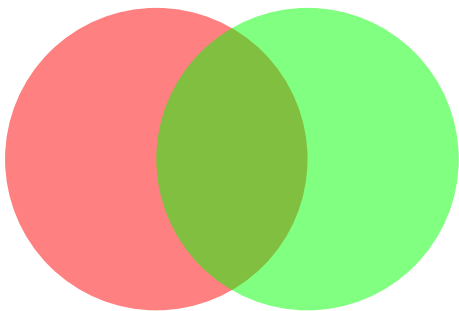
```
\raster[.5]{\blackrule[width=3cm,height=1cm]}\quad
\raster[.8]{\blackrule[width=3cm,height=1cm]}
```



Of course the real turture 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
```

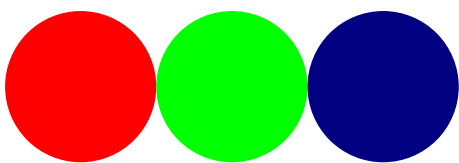
These transparent color circles up as:



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 is 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 T<sub>E</sub>X'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 T<sub>E</sub>X not having floats)

Although recent versions of PDF<sub>T</sub><sub>E</sub>X 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 package in 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 CONTEX 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{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{default {\sl slanted \bi bold italic} default}  
default {\bf bold \box0\ bold} default
```

Future versions of L<sup>A</sup>T<sub>E</sub>X 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 same method is used in MkII, but there 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 list 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 approaches.

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 un-  
set 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 un-  
set 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.*

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.

In `MkIV` spacing and linebreaks are dealt with by the analyser. Analysers are enabled by language switches but at some point I may decide to provide analysing independent of fonts.

```
\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] % hani kana
\definefontfeature
  [chinese-simple-hw]
  [mode=node,script=hang,lang=zhs] % hani kana

\definefontfeature
  [chinese-traditional-hw]
  [mode=node,script=hang,lang=zht,hwid=true,script=hani,lang=dflt]
\definefontfeature
  [chinese-simple-hw]
  [mode=node,script=hang,lang=zhs,hwid=true,script=hani,lang=dflt]

\definefontfeature
  [chinese-traditional-hw]
  [mode=node,hwid=true]
\definefontfeature
  [chinese-simple-hw]
  [mode=node,hwid=true]

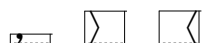
\font\ChinFont=name:adobesongstd-light*chinese-traditional-hw
```

traditional:

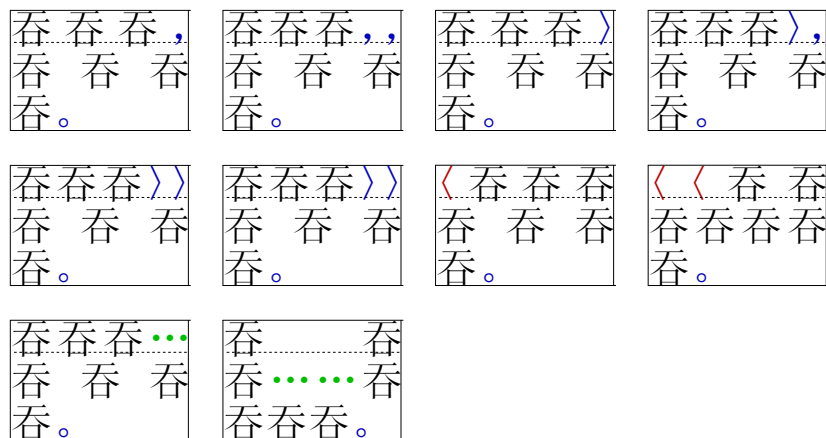
我〈能吞下玻璃而不傷身〉體。  
我〈能吞下玻璃而不傷身〉體。  
我〈能吞下玻璃而不傷身〉體。

simple:

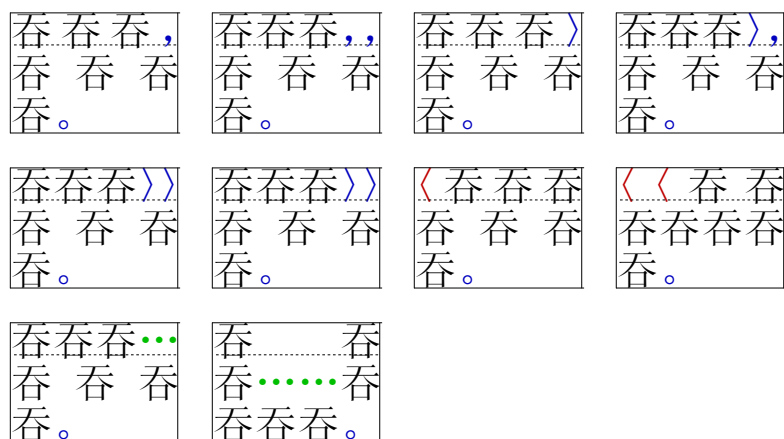
我〈能吞下玻璃而不伤身〉体。  
我〈能吞下玻璃而不伤身〉体。  
我〈能吞下玻璃而不伤身〉体。



### hsize 4.25em, fullwidth



### hsize 4.00em, fullwidth



### hsize 3.75em, fullwidth

吞 吞 吞 , 吞 吞 吞 吞。	吞 吞 吞 ,, 吞 吞 吞 吞。	吞 吞 吞 › 吞 吞 吞 吞。	吞 吞 吞 ›, 吞 吞 吞 吞。
吞 吞 吞 ›› 吞 吞 吞 吞。	吞 吞 吞 ›› 吞 吞 吞 吞。	‹ 吞 吞 吞 吞 吞 吞 吞。	‹‹ 吞 吞 吞 吞 吞 吞 吞吞。
吞 吞 吞 吞 … 吞 吞吞吞。	吞 吞 吞 吞 … … 吞 吞 吞 吞。		

### hsize 3.50em, fullwidth

吞 吞 吞 , 吞 吞 吞 吞。	吞 吞 吞 吞 , , 吞 吞吞吞。	吞 吞 吞 › 吞 吞 吞 吞。	吞 吞 吞 吞 › , 吞 吞吞吞。
吞 吞 吞 吞 › › 吞 吞吞吞。	吞 吞 吞 吞 › › 吞 吞吞吞。	‹ 吞 吞 吞 吞 吞 吞 吞。	‹‹ 吞 吞 吞 吞 吞 吞 吞吞。
吞 吞 吞 吞 … 吞 吞吞吞。	吞 吞 吞 吞 … … 吞 吞 吞 吞。		

### hsize 3.25em, fullwidth

吞 吞 吞 , 吞 吞 吞 吞。	吞 吞 吞 , , 吞 吞 吞 吞。	吞 吞 吞 › 吞 吞 吞 吞。	吞 吞 吞 › , 吞 吞 吞 吞。
吞 吞 吞 › › 吞 吞 吞 吞。	吞 吞 吞 › › 吞 吞 吞 吞。	‹ 吞 吞 吞 吞 吞 吞吞。	‹‹ 吞 吞 吞 吞 吞 吞吞。

吞	吞
吞	...
吞	吞
吞	吞
吞	吞

### hsize 3.00em, fullwidth

吞	吞
吞	,
吞	吞
吞	吞
吞	吞

吞	吞
吞	,
吞	吞
吞	吞
吞	吞

吞	吞
吞	>
吞	吞
吞	吞
吞	吞

吞	吞
吞	>
吞	,
吞	吞
吞	吞

吞	吞
吞	>
吞	吞
吞	吞
吞	吞

吞	吞
吞	>
吞	吞
吞	吞
吞	吞

<	吞
吞	吞
吞	吞
吞	吞
吞	吞

<<	吞
吞	吞
吞	吞
吞	吞
吞	吞

吞	吞
吞	...
吞	吞
吞	吞
吞	吞

吞	吞
吞	...
吞	吞
吞	吞
吞	吞

### hsize 4.25em, halfwidth

吞	吞	吞
吞	吞	,
吞	吞	吞
吞	吞	吞
吞	吞	吞

吞	吞	吞
吞	吞	‘
吞	吞	吞
吞	吞	吞
吞	吞	吞

吞	吞	吞
吞	吞	’
吞	吞	吞
吞	吞	吞
吞	吞	吞

### hsize 4.00em, halfwidth

吞	吞	吞
吞	吞	,
吞	吞	吞
吞	吞	吞
吞	吞	吞

吞	吞	吞
吞	吞	‘
吞	吞	吞
吞	吞	吞
吞	吞	吞

吞	吞	吞
吞	吞	’
吞	吞	吞
吞	吞	吞
吞	吞	吞

### hsize 3.75em, halfwidth

吞	吞	吞
吞	吞	,
吞	吞	吞
吞	吞	吞
吞	吞	吞

吞	吞	吞
吞	吞	‘
吞	吞	吞
吞	吞	吞
吞	吞	吞

吞	吞	吞
吞	吞	’
吞	吞	吞
吞	吞	吞
吞	吞	吞



### hsize 3.50em, halfwidth

吞吞吞,	吞吞吞	吞吞吞
吞吞吞	‘吞吞	吞吞吞
吞。	吞吞。	吞吞吞。

### hsize 3.25em, halfwidth

吞吞吞,	吞吞吞	吞吞吞
吞吞吞	‘吞吞	吞吞吞
吞。	吞吞。	吞吞吞。

### hsize 3.00em, halfwidth

吞吞吞,	吞吞吞	吞吞吞
吞吞吞	‘吞吞	吞吞吞
吞。	吞吞。	吞吞吞。



## 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 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
```

```
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:

acticity	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 no 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, it 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 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, efficiency 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 Ierusalimsky 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 T<sub>E</sub>X-Gyre fonts. The troubles originate in the fact that the OPEN<sub>T</sub>YPE 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 T<sub>E</sub>X 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
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

*under construction*

### the parser

For quite a while CON<sub>T</sub>EX<sub>T</sub> 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 LUA<sub>T</sub>EX 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 CON<sub>T</sub>EX<sub>T</sub>):

```
\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
    tex.sprint("\\starttyping")
    xml.serialize(document.xml, tex.sprint)
    tex.sprint("\\stoptyping")
\stopluacode
```

We can control the way the serializer deals with the snippets, here we just print back to T<sub>E</sub>X.

```
<?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>

```

We can also pass a third argument:

```

\startluacode
    tex.sprint("\\starttyping")
    xml.serialize(document.xml, tex.sprint, string.upper, string.upper)
    tex.sprint("\\stoptyping")
\stopluacode

```

This returns:

```

<?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>LUALEX</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 T<sub>E</sub>X reading from file.

```

\startluacode
  document.str = "<l> <w>hello</w> <w>world</w> </l>"
  tex.sprint("\\starttyping")
  xml.serialize(xml.convert(document.str),tex.sprint)
  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>

```

An optional second argument of the converter determines if we deal with a root element.

```

\startluacode
  tex.sprint("\\starttyping")
  xml.serialize(xml.convert(document.str,false),tex.sprint)
  tex.sprint("\\stoptyping")
\stopluacode

```

Now we get this:

```

<l> <w>hello</w> <w>world</w> </l>

```

You can save a (manipulated) XML table with the command:

```

\startluacode
  xml.save(document.xml,"newfile.xml")
\stopluacode

```

These examples show that you can manipulate 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.

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>

```
\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>xetex</some>

```
\startluacode
    xml.sprint(xml.filter(document.xml, "/one/three/some/texts()"))
\stopluacode
```

result: pdftexluatexxetex

```
\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 pre-processing (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.

## 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.

pattern	supported	comment
<b>a</b>	★	not anchored
<b>!a</b>	★	not anchored,negated
<b>a/b</b>	★	anchored on preceding
<b>/a/b</b>	★	anchored (current root)
<b>^a/c</b>	★	anchored (current root)
<b>^^/a/c</b>	todo	anchored (document root)
<b>a/*/b</b>	★	one wildcard
<b>a//b</b>	★	many wildcards
<b>a/**/b</b>	★	many wildcards
<b>.</b>	★	ignored self
<b>..</b>	★	parent

<code>a[5]</code>	*	index upwards
<code>a[-5]</code>	*	index downwards
<code>a[position()=5]</code>	maybe	
<code>a[first()]</code>	maybe	
<code>a[last()]</code>	maybe	
<code>(b c d)</code>	*	alternates (one of)
<code>b c d</code>	*	alternates (one of)
<code>!(b c d)</code>	*	not one of
<code>a/(b c d)/e/f</code>	*	anchored alternates
<code>(c/d e)</code>	not likely	nested subpaths
<code>a/b[@bla]</code>	*	any value of
<code>a/b/@bla</code>	*	any value of
<code>a/b[@bla='oeps']</code>	*	equals value
<code>a/b[@bla=='oeps']</code>	*	equals value
<code>a/b[@bla&lt;&gt;'oeps']</code>	*	different value
<code>a/b[@bla!='oeps']</code>	*	different value
<code>...../attribute(id)</code>	*	
<code>...../attributes()</code>	*	
<code>...../text()</code>	*	
<code>...../texts()</code>	*	
<code>...../first()</code>	*	
<code>...../last()</code>	*	
<code>...../index(n)</code>	*	
<code>...../position(n)</code>	*	
<code>root::</code>	*	
<code>parent::</code>	*	
<code>child::</code>	*	
<code>ancestor::</code>	*	
<code>preceding-sibling::</code>	not soon	
<code>following-sibling::</code>	not soon	
<code>preceding-sibling-of-self::</code>	not soon	
<code>following-sibling-or-self::</code>	not soon	
<code>descendent::</code>	not soon	
<code>preceding::</code>	not soon	
<code>following::</code>	not soon	
<code>self::node()</code>	not soon	
<code>id("tag")</code>	not soon	
<code>node()</code>	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
    tex.sprint(xml.join(xml.collect_texts(
        document.xml, "/one/(three|five)/some"
    ), ', ', ' and '))
\stopluacode

```

We get: ‘pdf<sub>te</sub>x, lua<sub>te</sub>x, xet<sub>ex</sub> and meta<sub>pos</sub>t’.

There are several helper functions, like `xml.count` which in this case returns 4.

```

\startluacode
    tex.sprint(xml.count(document.xml, "/one/(three|five)/some"))
\stopluacode

```

Functions like this give 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, "/one/three/some", "<be>okay</be>")
    xml.after (document.xml, "/one/three/some", "<af>okay</af>")
    tex.sprint("\starttyping")
    xml.serialize_path(document.xml, "/one/three", tex.sprint)
    tex.sprint("\stoptyping")
\stopluacode

```

And indeed, we suddenly have a couple of ‘okay’s there:

```

<three>
    <be>okay</be><some>pdftex</some><af>okay</af>
    <be>okay</be><some>luatex</some><af>okay</af>
    <be>okay</be><some>xetex</some><af>okay</af>
</three>

```

Of course we can also delete elements:

```

\startluacode
    xml.delete(document.xml, "/one/three/some")
    xml.delete(document.xml, "/one/three/af")
    tex.sprint("\starttyping")
    xml.serialize_path(document.xml, "/one/three", tex.sprint)

```

```
tex.sprint("\\stoptyping")
\stopluacode
```

Now we have:

```
<three>
  <be>okay</be>
  <be>okay</be>
  <be>okay</be>
</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.serialize_path(document.xml, "/one/three", tex.sprint)
  tex.sprint("\\stoptyping")
\stopluacode
```

And indeed we get:

```
<three>
  <mid>done</mid>
  <mid>done</mid>
  <mid>done</mid>
</three>
```

These are just a few features of the library. I will add some more (rather) generic manipulators 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 T<sub>E</sub>X (most of the XML that I deal with is rather dirty and needs some cleanup).

## streaming trees

Eventually we will provide 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 and 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 T<sub>E</sub>X 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 T<sub>E</sub>X is also efficient. Since we no longer need to manipulate the elements in T<sub>E</sub>X 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
<code>\xmlfirst</code>	<code>{main} {/one/three/some}</code>	<code>&lt;some&gt;pdfTeX&lt;/some&gt;</code>
<code>\xmllast</code>	<code>{main} {/one/three/some}</code>	<code>&lt;some&gt;xetex&lt;/some&gt;</code>
<code>\xmlindex</code>	<code>{main} {/one/three/some} {2}</code>	<code>&lt;some&gt;luatex&lt;/some&gt;</code>

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 is takes the same runtime for both approached. 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 T<sub>E</sub>X 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><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 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><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><d>d3</d></c>

<c n="y"><d>d4</d></c>

<c><d>d5</d></c>

**c[@n='y']**

<c n="y">c2</c>

<c n="y"><d>d4</d></c>

**c[1]**

```
<c n="x">c1</c>
<c n="y">c2</c>
<c><d>d3</d></c>
```

**b/c[1]**

```
<c n="x">c1</c>
<c n="y">c2</c>
```

**a/c[1]**

```
<c><d>d3</d></c>
```

**a/c[-1]**

```
<c><d>d5</d></c>
```

**c[1]**

```
<c n="x">c1</c>
<c n="y">c2</c>
<c><d>d3</d></c>
```

**c[-1]**

```
<c><d>d5</d></c>
```

**pi::**

```
<?xml version='1.0' ?>
<?what is this?>
<?what is that?>
```

**pi::what**

```
<?what is this?>
<?what is that?>
```

## 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 T<sub>E</sub>X: 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<sup>1</sup>

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 powder. 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, T<sub>E</sub>X 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.

---

<sup>1</sup> The World Without Us, Alan Weisman; a quote from Richard Thomson in chapter: Polymers are Forever.

The code in traditional T<sub>E</sub>X that deals with hyphenation and linebreaks is rather interwoven. There is a relationship between the font encoding and the way patterns are encoded. A few years after T<sub>E</sub>X 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 T<sub>E</sub>X 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 CON<sub>T</sub>E<sub>X</sub>T, 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 M<sub>K</sub>IV 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 L<sub>U</sub>A<sub>T</sub>E<sub>X</sub> 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 (L<sub>U</sub>A could not handle the large tables as function return value) and I made some hyphenation code in L<sub>U</sub>A. 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 T<sub>E</sub>X 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 exercise 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 OPEN<sub>T</sub>YPE 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 T<sub>E</sub>X 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 T<sub>E</sub>X is global (well, in L<sub>U</sub>A<sub>T</sub><sub>E</sub>X less and less) this complicates matters. Because in MkIV hyphenation, ligature building and kerning are done differently (due to OPEN<sub>T</sub>YPE) 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 definitely be qualified as an ugly hack.

Before we discuss how this was solved, we summarize the state of affairs. In L<sub>U</sub>A<sub>T</sub><sub>E</sub>X 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 its 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 T<sub>E</sub>X does not look at all potential hyphenation points, but only around places that have a high probability as line-end. L<sup>A</sup>T<sub>E</sub>X 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  $\varepsilon$ -T<sub>E</sub>X extension). The usual left- and righthyphen-min 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 T<sub>E</sub>X, 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 and 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 unboxed example, the unboxed 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 unboxed, 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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X, 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 discretionary 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 occurred. 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.

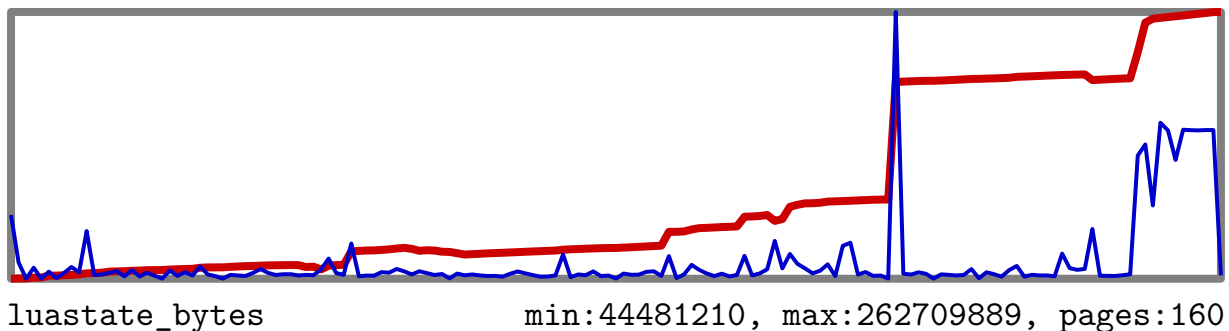
```
con-text == [c] [o] (pre=n-,post=,replace=n) [t] [e] [x] [t]  
effe     == [e] (pre=f-,post=f,replace=ff) [e]
```

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 take 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

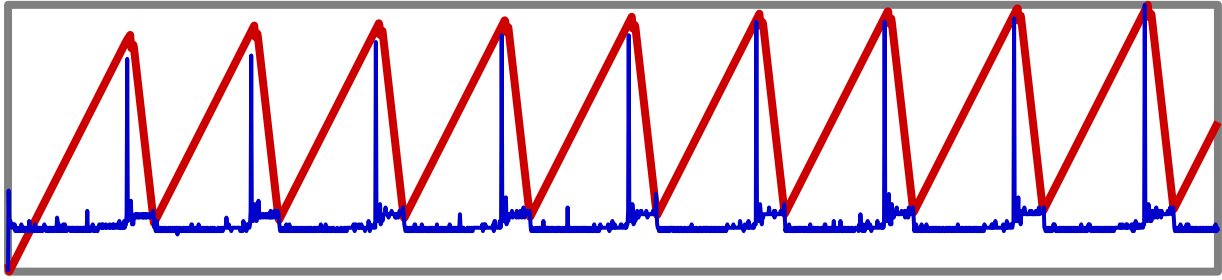
We use the `mk.tex` document for testing and because it keeps track of how L<sup>A</sup>T<sub>E</sub>X evolves. As a result it has some uncommon characteristics. For instance, you can see increments in memory usage at points where we load fonts: the chapters on Zapfino, Arabic and CJK (unfinished). This memory is not freed because the font memory is used permanently. In the following graphic, the red line is the memory consumption of L<sup>A</sup>T<sub>E</sub>X for the current version of `mk.tex`. The blue line is the runtime per page.



At the moment of writing this Taco has optimized the L<sup>A</sup>T<sub>E</sub>X 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 OPEN<sub>T</sub>YPE 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.

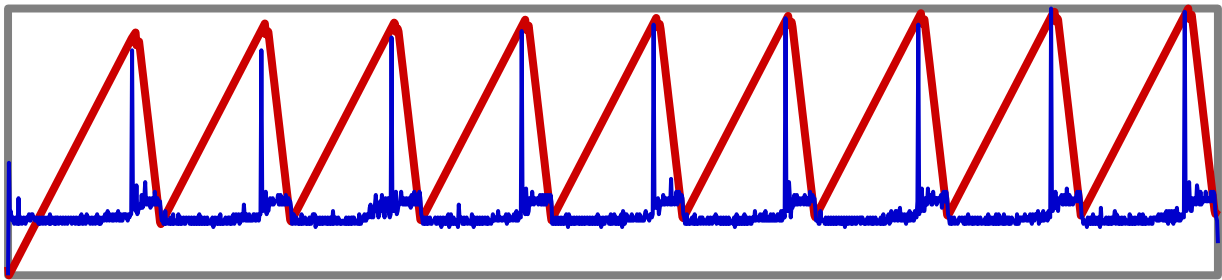
```
\usemodule[timing] \starttext \dorecurse{2000}{  
  \input tufte \par \input tufte \par \input tufte \page  
} \stoptext
```



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.

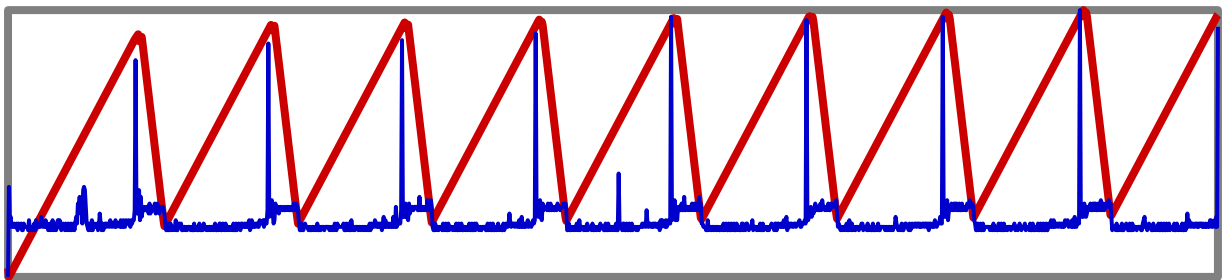
```
\usemodule[timing] \starttext \dorecurse{2000}{
  \input tufte \par \input tufte \par \input tufte \par
} \stoptext
```



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 much 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
```

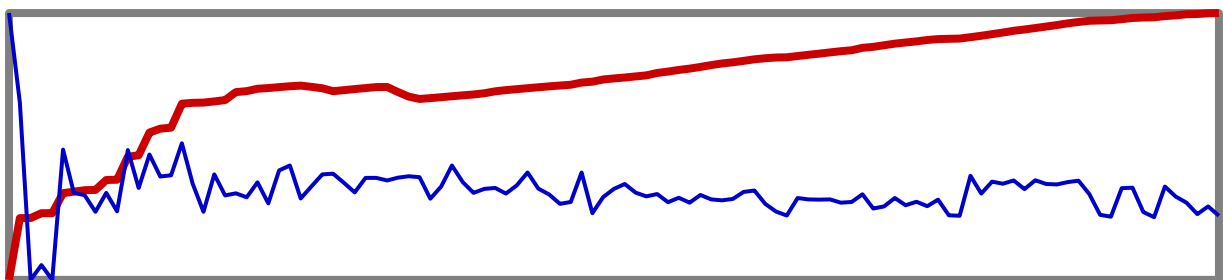


luastate\_bytes min:36876892, max:86359763, pages:1385

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).

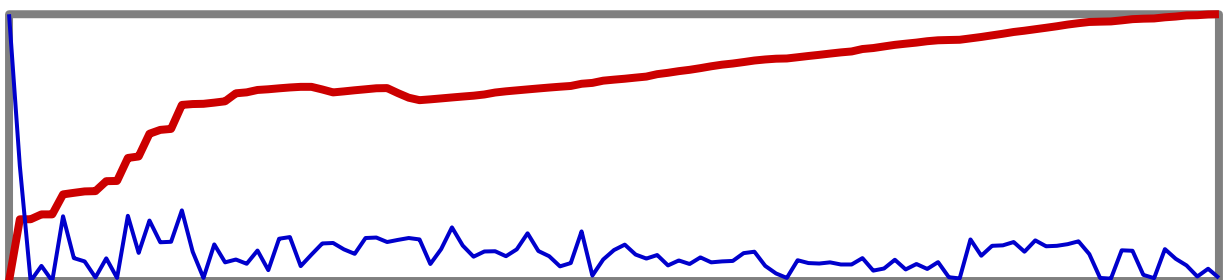
Keep in mind that these quotes in `tufte.tex` are just test samples, and not that realistic in everyday documents. On the other hand, these tests involve the usual font loading, node processing, attribute handling etc. They provide a decent baseline.

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 pagnumber 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 influenced 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.

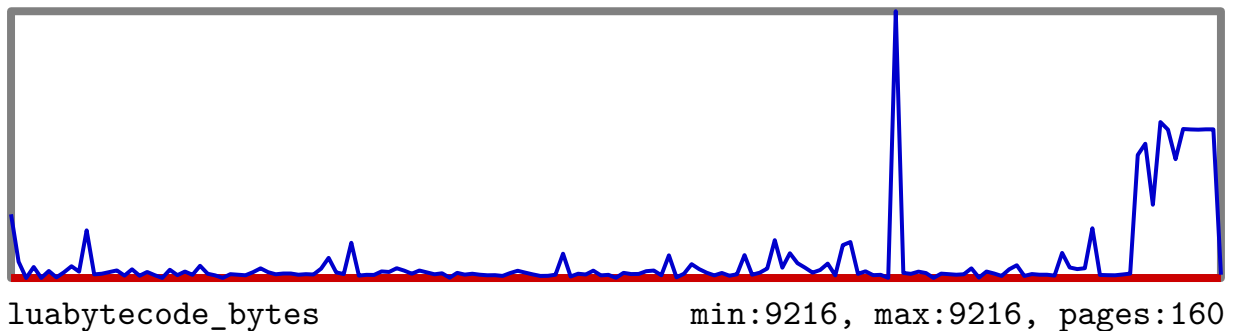
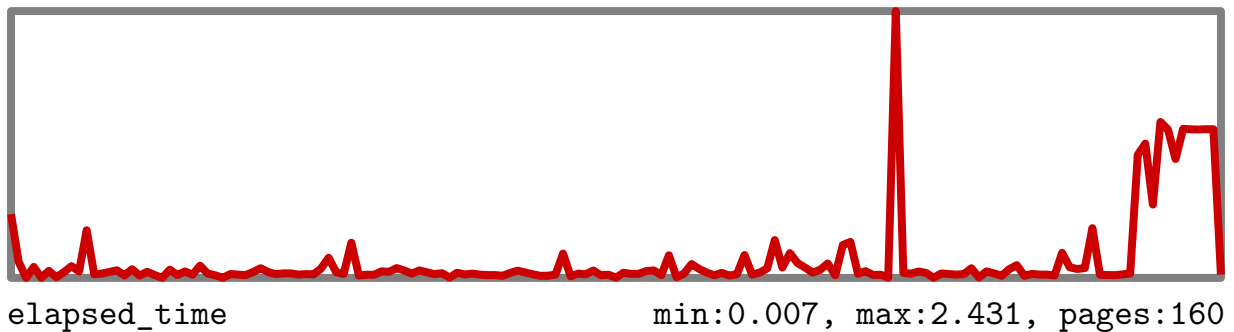
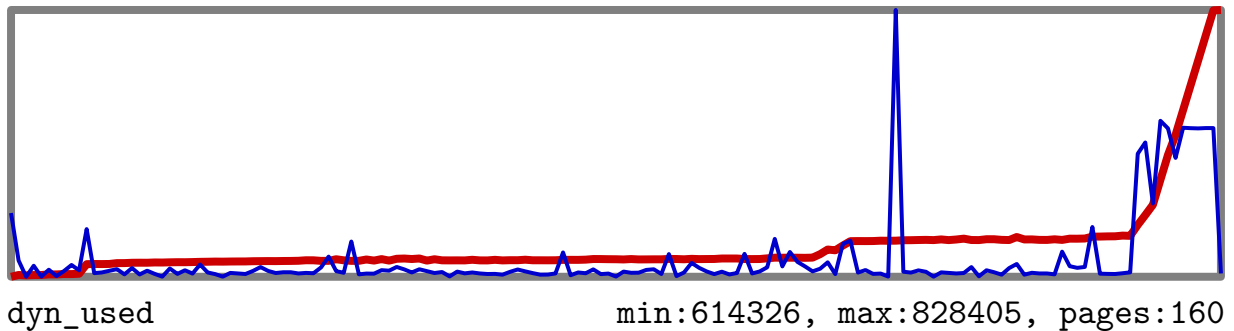
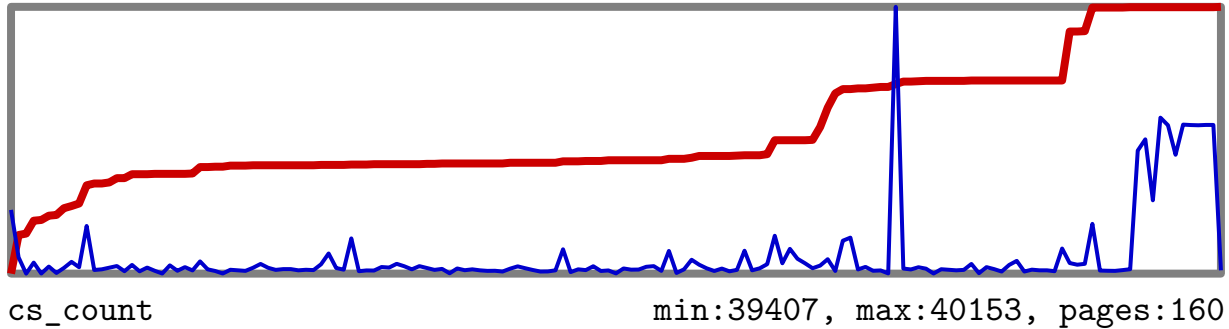


luastate\_bytes min:59690749, max:155669371, pages:112

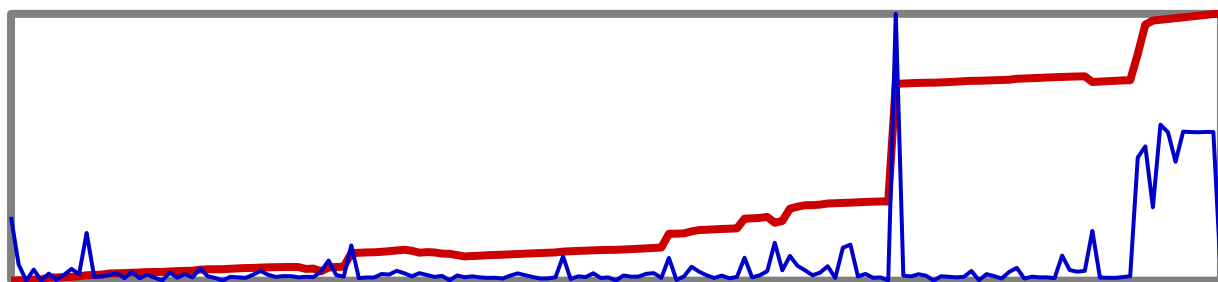
In the `mk.tex` document we use TYPE1 fonts for the main body of the text and load some (huge) OPENTYPE fonts later on. Here we use OPENTYPE fonts exclusively and since CONTeXt loads fonts only when needed, you see several spikes in the time per page bars and memory consumption quickly becomes stable. 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.

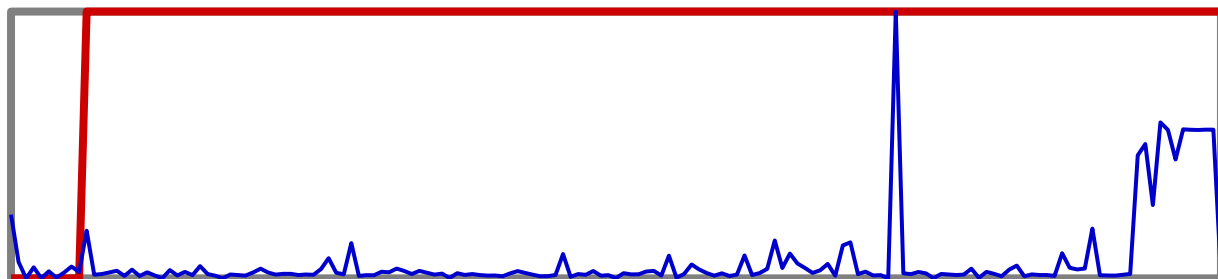
The previous graphics combine LUA memory consumption with time spent per page. The following graphics show variants of this. The graphics concern this document (`mk.tex`). Again, the blue lines represent the runtime per page.



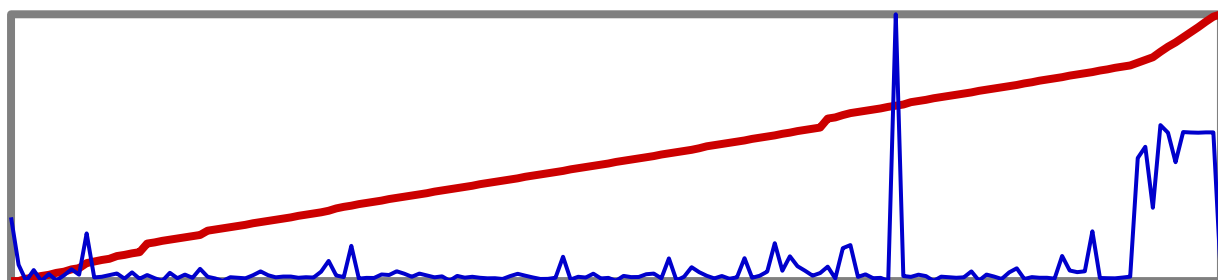




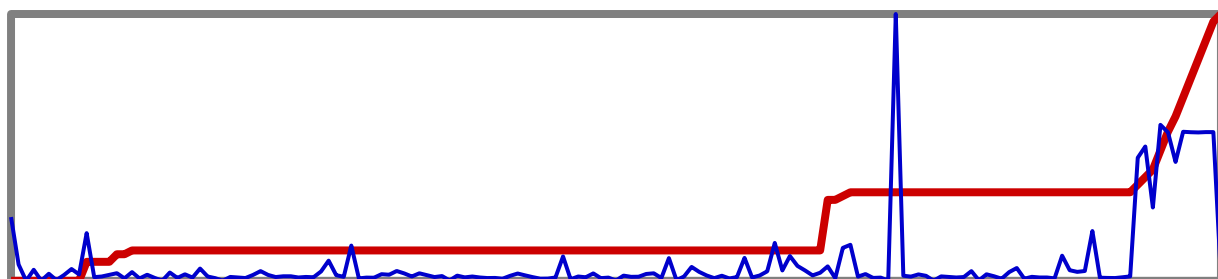
luastate\_bytes min:44481210, max:262709889, pages:160



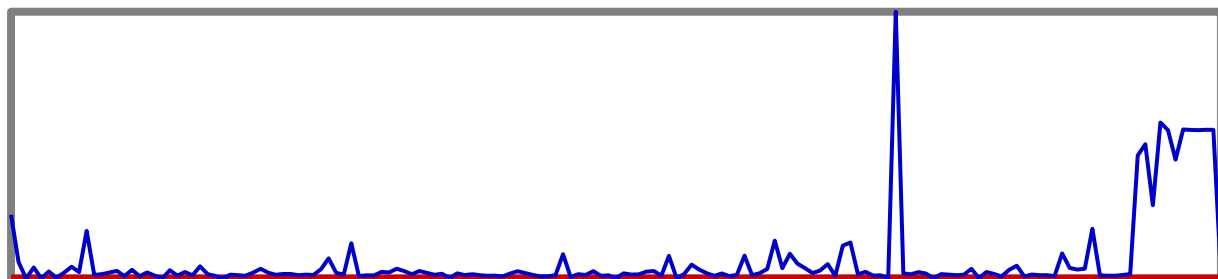
max\_buf\_stack min:254, max:369, pages:160



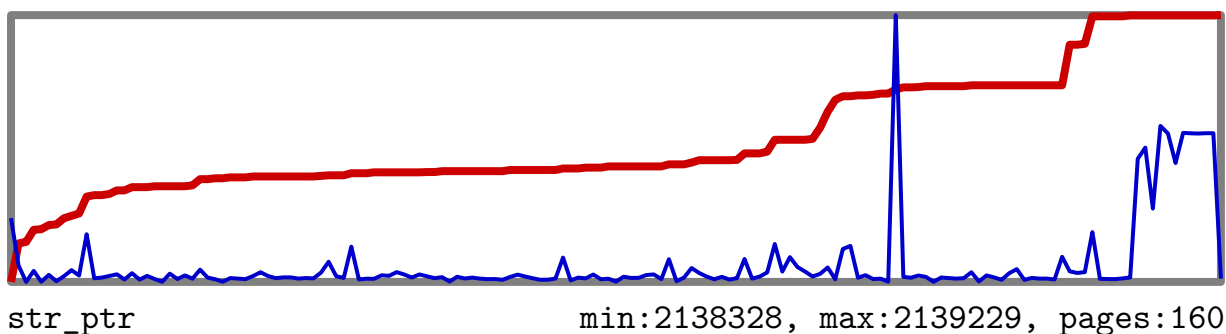
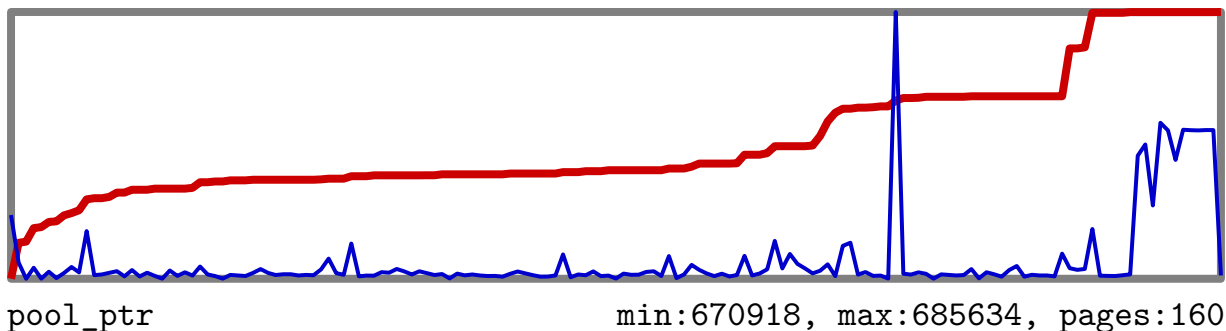
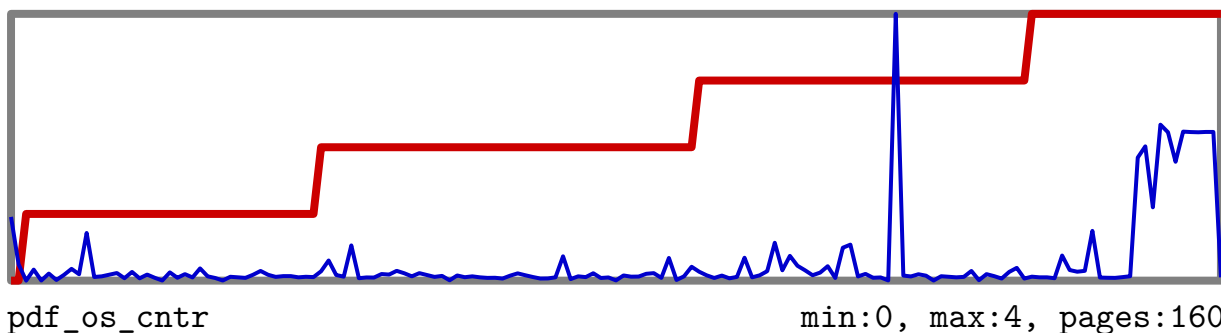
obj\_ptr min:0, max:683, pages:160



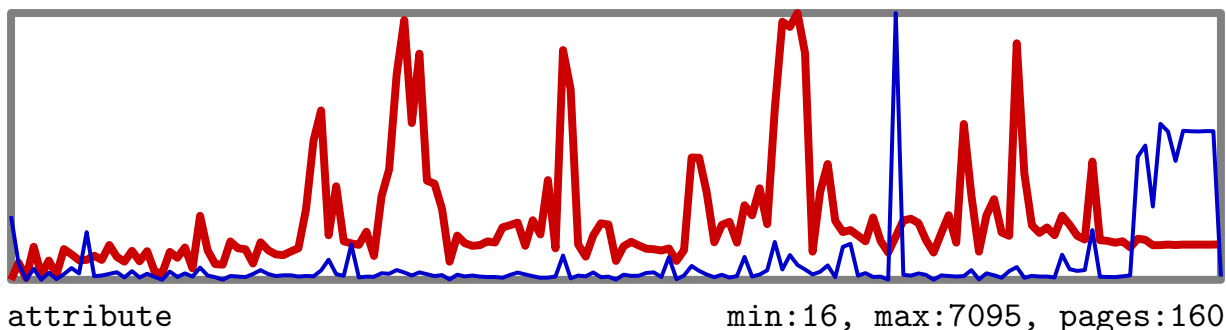
pdf\_mem\_ptr min:1, max:423, pages:160

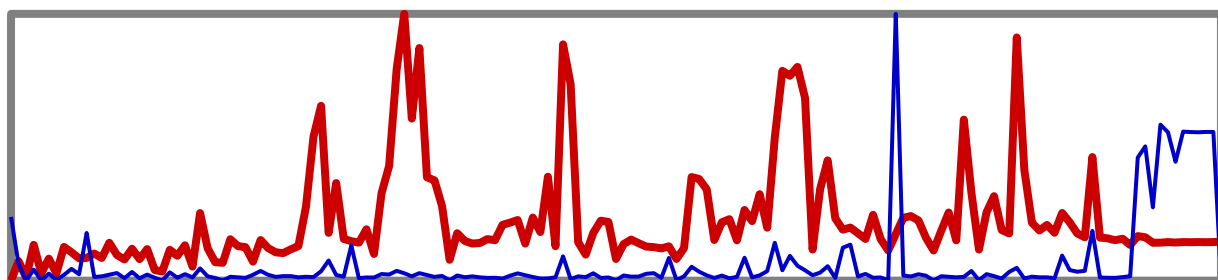


pdf\_mem\_size min:10000, max:10000, pages:160



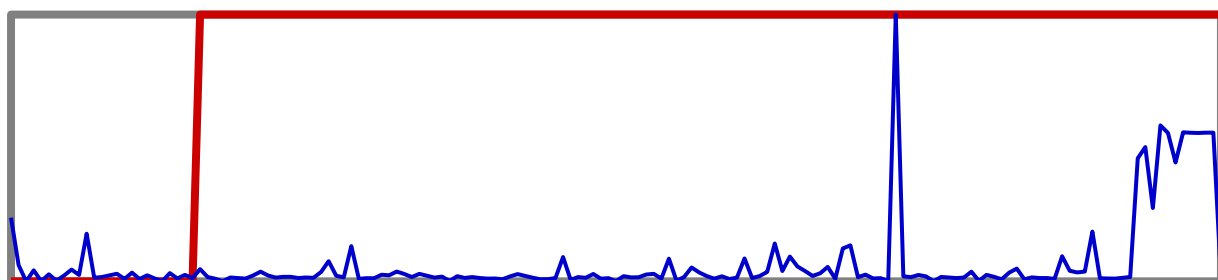
In L<sup>A</sup>T<sub>E</sub>X node memory management is rewritten. Contrary to what you may expect, node memory consumption is not that large. Pages seldom contain more than 5000 nodes, although extensive use of attributes can easily duplicate this. Node usage in this documents is as follows.





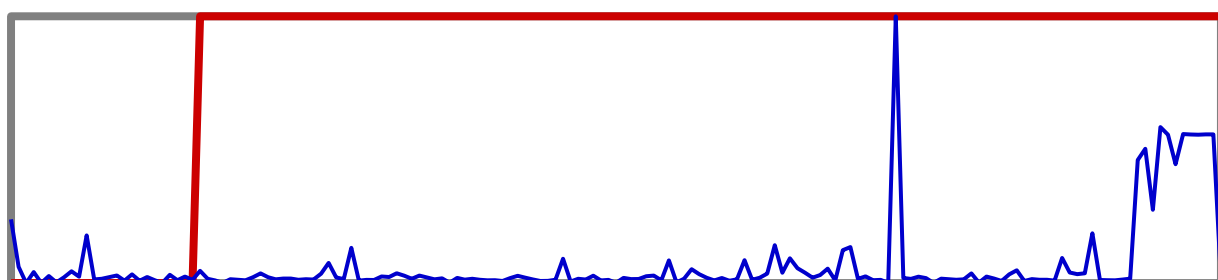
attribute\_list

min:8, max:2326, pages:160



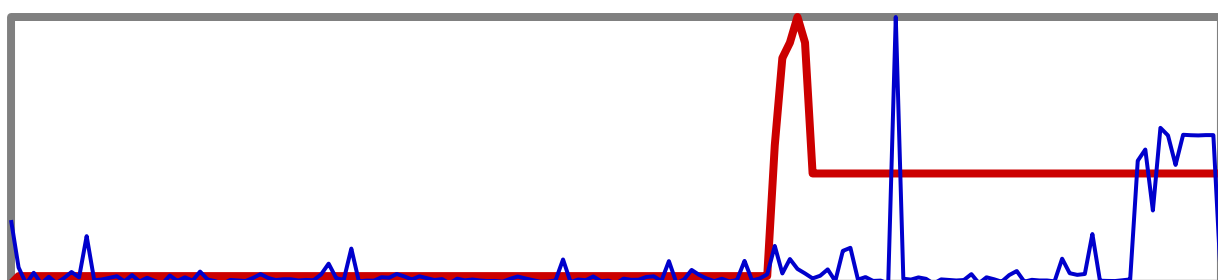
bin

min:0, max:48, pages:160



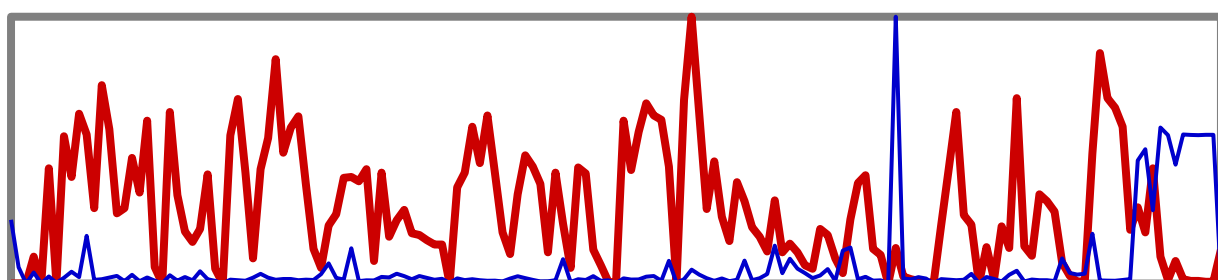
choice

min:0, max:12, pages:160



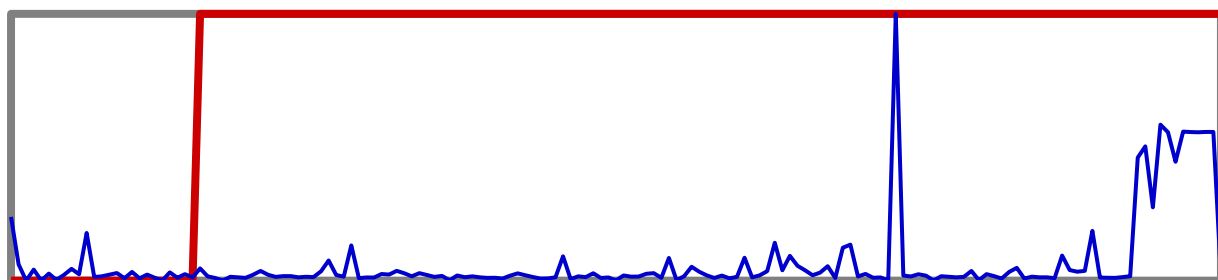
dir

min:2, max:106, pages:160



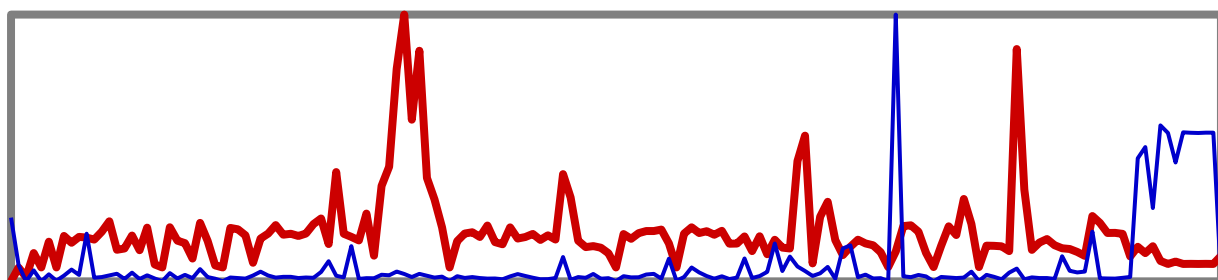
disc

min:1, max:309, pages:160



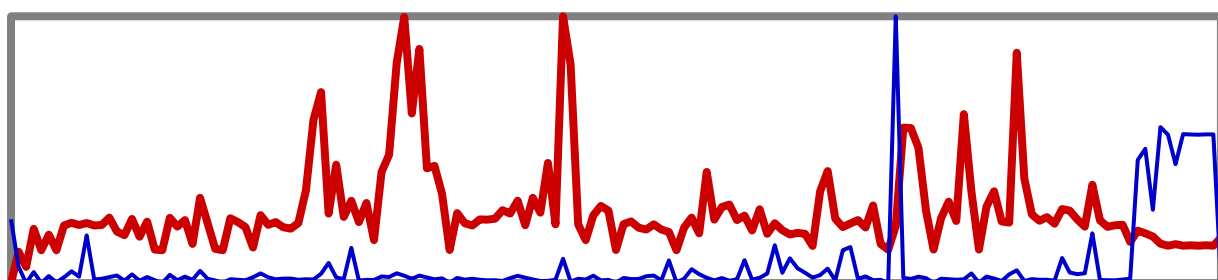
fraction

min:0, max:6, pages:160



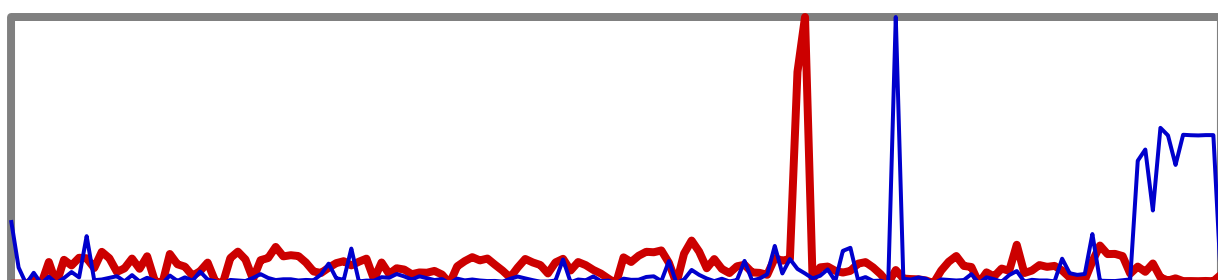
glue

min:1, max:4800, pages:160



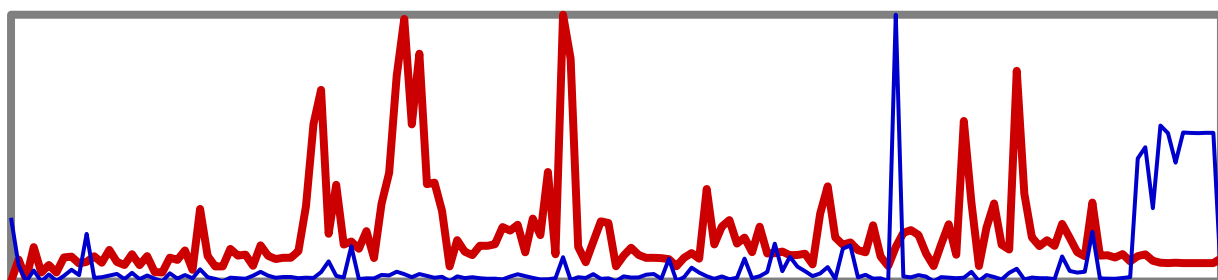
glue\_spec

min:19, max:1205, pages:160



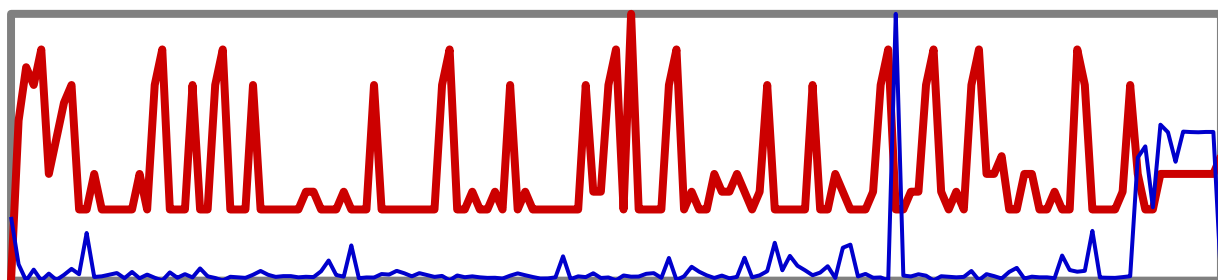
glyph

min:0, max:25141, pages:160



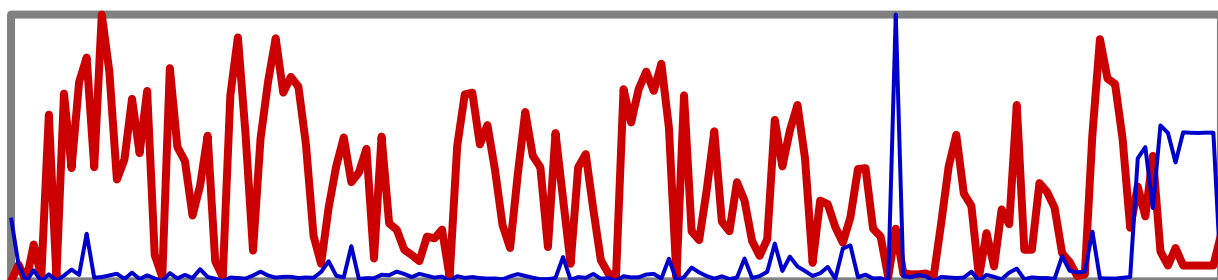
hlist

min:3, max:2105, pages:160



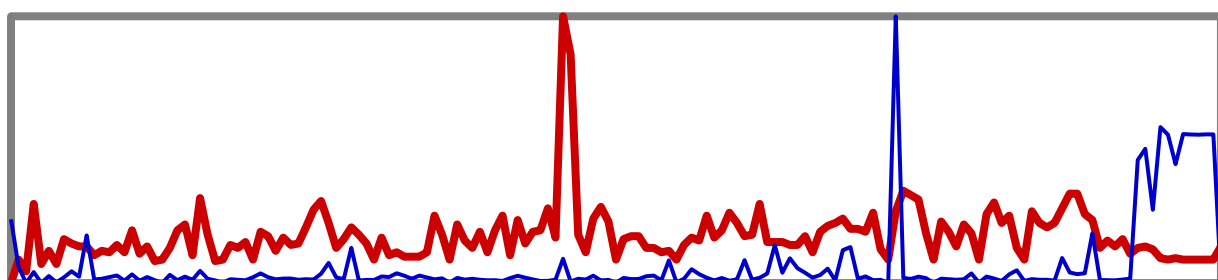
if\_stack

min:0, max:15, pages:160



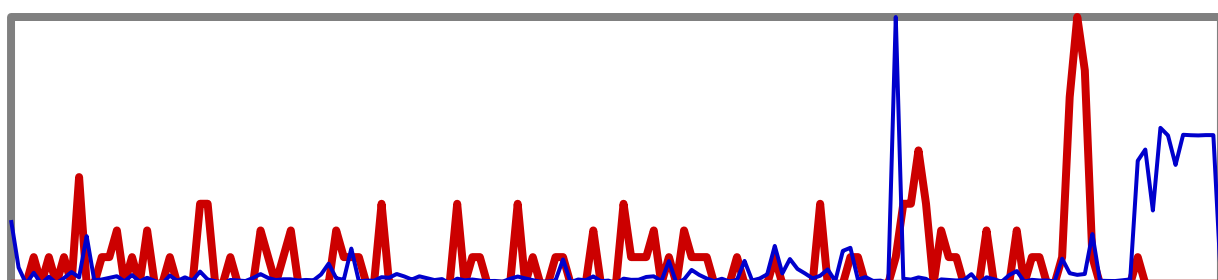
kern

min:1, max:305, pages:160



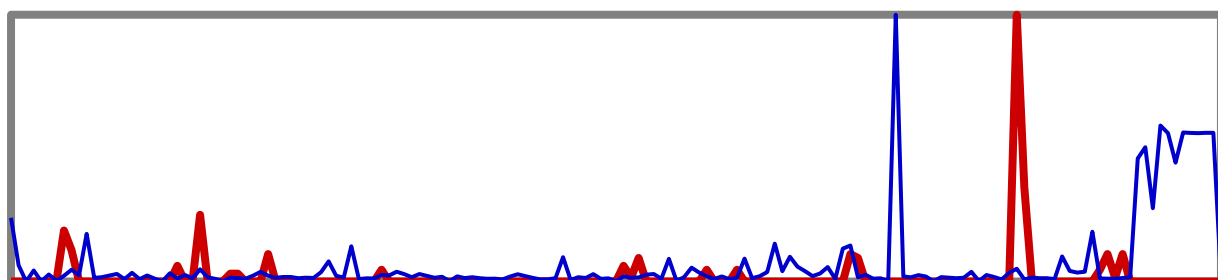
local\_par

min:0, max:182, pages:160



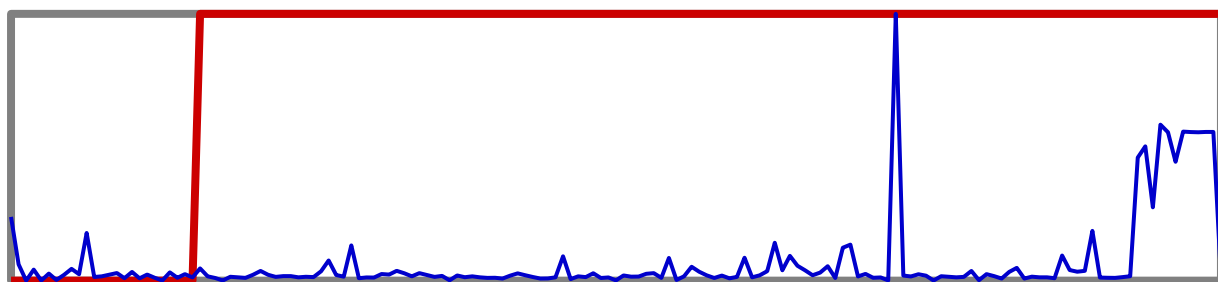
mark

min:0, max:40, pages:160



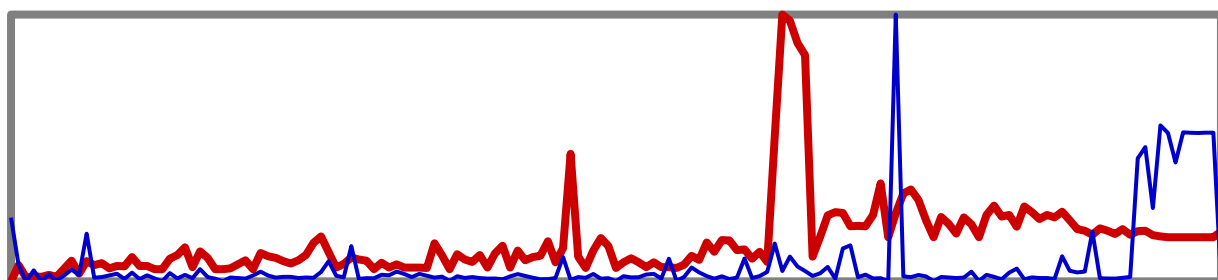
math

min:0, max:136, pages:160



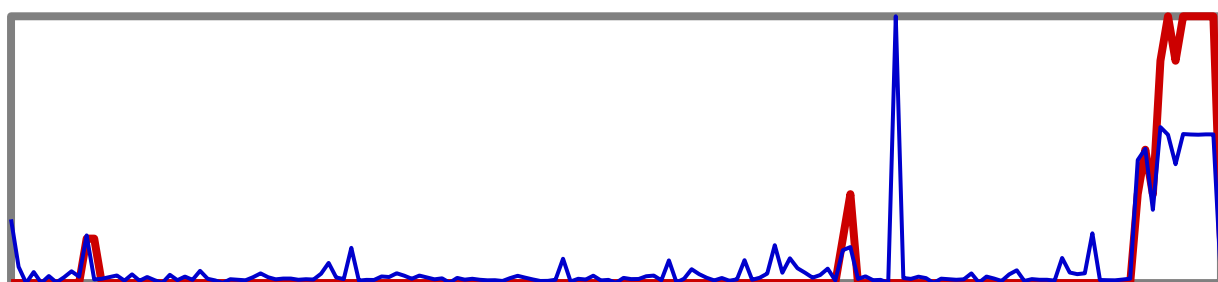
ord

min:0, max:108, pages:160



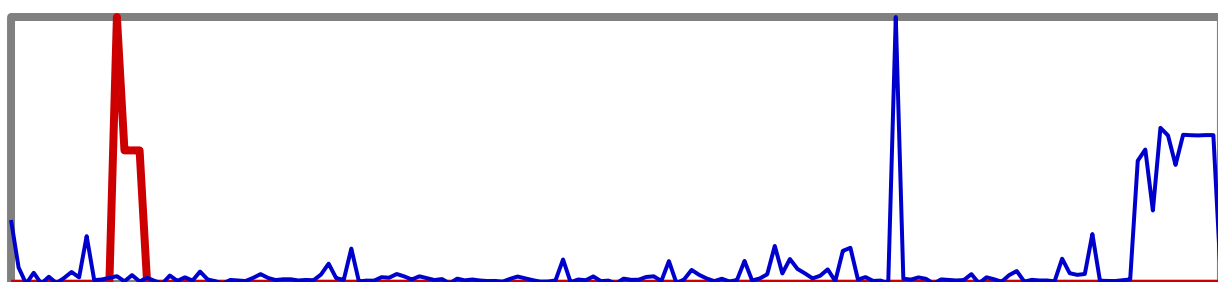
pdf\_literal

min:29, max:688, pages:160



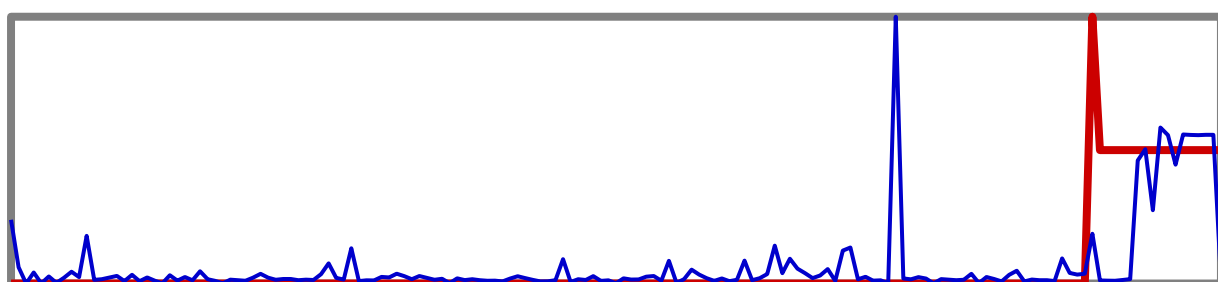
pdf\_refxform

min:0, max:6, pages:160



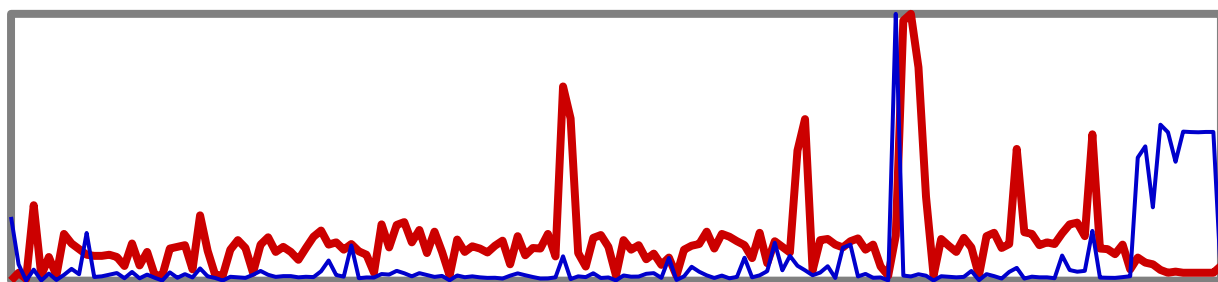
pdf\_refximage

min:0, max:2, pages:160



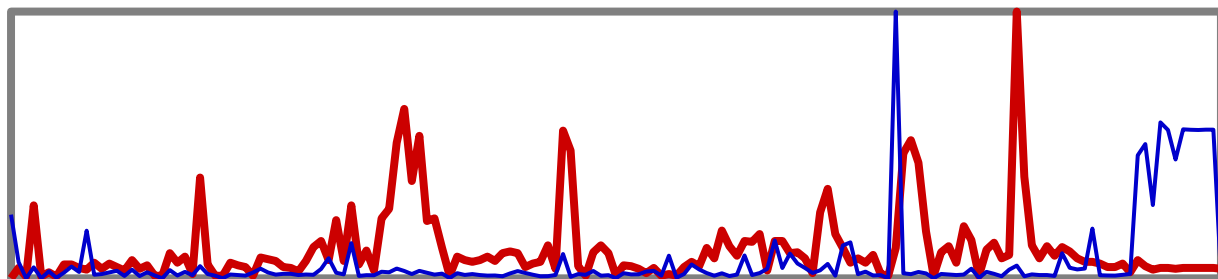
pdf\_save\_pos

min:0, max:2, pages:160



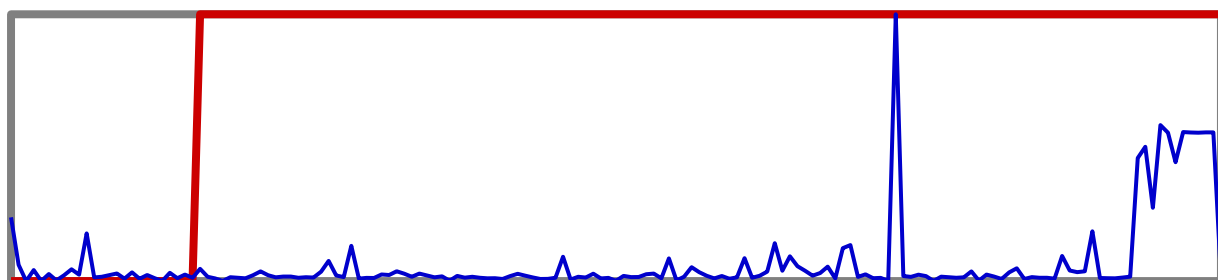
penalty

min:1, max:475, pages:160



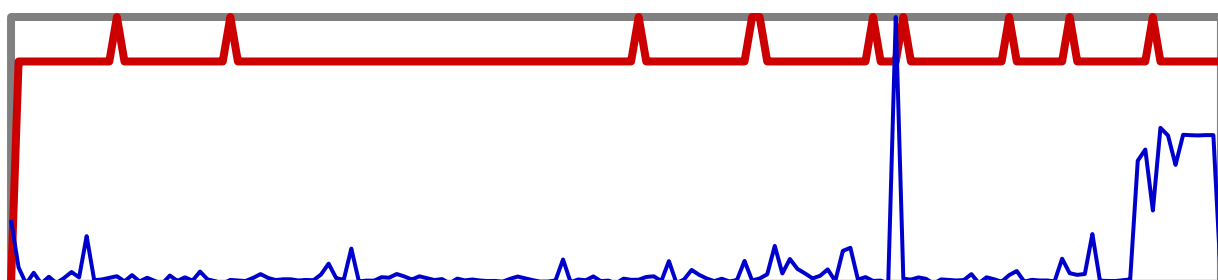
rule

min:2, max:309, pages:160



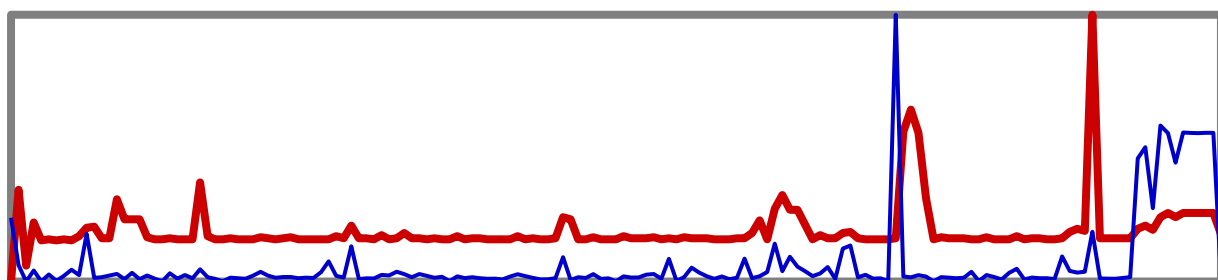
style

min:0, max:48, pages:160



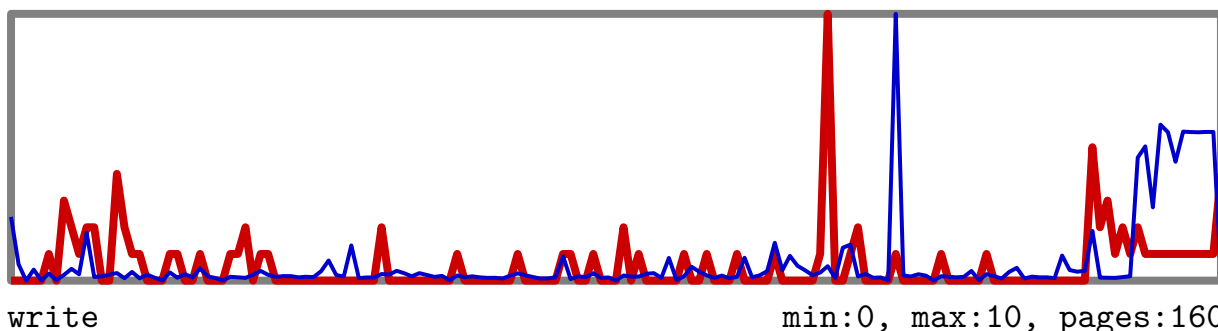
temp

min:0, max:6, pages:160



vlist

min:5, max:258, pages:160



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.)