MkIV Hybrid Technology

Contents

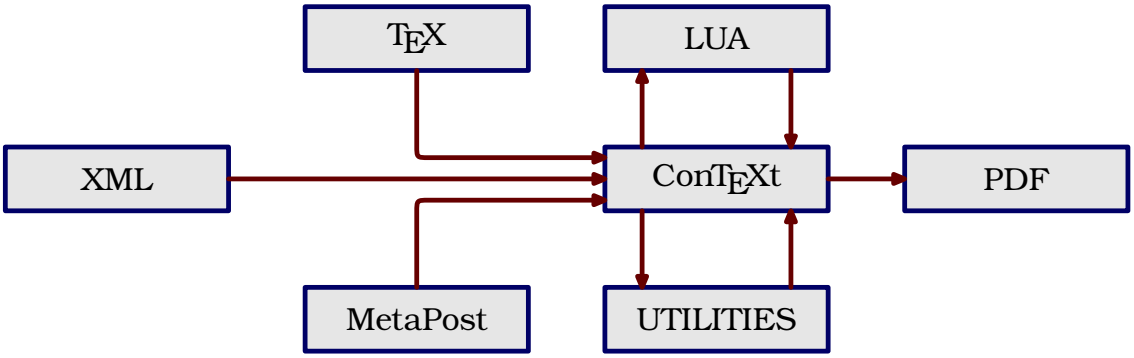
Introduction

We're halfway the development of LuaTEX (mid 2009) and substantial parts of ConTEXt have been rewritten using a mixture of Lua and TEX. In another document, "ConTEXt MkII–MkIV, the history of LuaTEX 2006–2009", we have kept track of how both systems evolved so far[1]. Here we continue that story which eventually will end with both systems being stable and more of less complete in their basic features.

The title of this document needs some explanation, although the symbols on the cover might give a clue already. In ConTEXt MkIV, as it is now, we mix several languages:

- good old TEX: here you will see `{}` all over the place
- fancy MetaPost: there we use quite some `()`
- lean and mean Lua: both `{}` and `()` show up a lot there
- unreadable but handy xml: immediately recognizable by the use of `<>`

As we use all of them mixed, you can consider MkIV to be a hybrid system and just as with hybrid cars, efficiency is part of the concept.



In this graphic we've given Lua a somewhat different place than the other three languages. First of all we have Lua inside TEX, which is kind of hidden, but at the same time we can use Lua to provide whatever extra features we need, especially when we've reached the state where we can load libraries. In a similar fashion we have utilities (now all written in Lua) that can manage your workflow or aspects of a run (the `mtxrun` script plays a central role in this).

[1] Parts of this have been published in usergroup magazines like the Maps, TugBoat, and conference proceedings of EuroTEX and tug.

The mentioned history document was (and still is) a rather good testcase for LuaTeX and MkIV. We explore some new features and load a lot of fonts, some really large. This document will also serve that purpose. This is one of the reasons why we have turned on grid snapping (and occasionally some tracing).

Keeping track of the history of LuaTeX and MkIV in a document serves several purposes. Of course it shows what has been done. It also serves as a reminder of why it was done that way. As mentioned it serves as test, both in functionality and performance, and as such it's always one of the first documents we run after a change in the code. Most of all this document serves as an extension to my limited memory. When I look at my source code I often can remember when and why it was done that way at that time. However, writing it down more explicitly helps me to remember more and might help users to get some insight in the developments and decisions made.[2]

A couple of new features were added to LuaTeX in 2010 but the years 2011 and 2012 were mostly spent on fixing issues and reaching a stable state. In parallel parts of ConTeXt were rewritten using Lua and new possibilities have been explored. Indeed LuaTeX had become pretty stable, especially because we used it in production. There are still a lot of things on the agenda but with LuaTeX 0.75 we have reached yet another milestone: integration of Lua 5.2, exploration of LuaJIT, and the move forward to a version of MetaPost that supports doubles as numeric type. Luigi Scarso and I also started the SwigLib project that will make the use of external libraries more easy.

Of course, although I wrote most of the text, this document is as much a reflection of what Taco Hoekwater and Hartmut Henkel come up with, but all errors you find here are definitely mine. Some chapters have been published in TugBoat, the Maps and other usergroup journals. Some chapters have become manuals, like the one on spreadsheets. I also owe thanks to the ConTeXt community and those active on the mailing list: it's a real pleasure to see how fast new features are picked up and how willing to test users are when new betas show up.

Hans Hagen, Hasselt NL,
September 2009 — December 2012

http://www.luatex.org
http://www.pragma-ade.com

[2] I read a lot and regret that I forget most of what I read so fast. I might as well forget what I wrote so have some patience with me as I repeat myself occasionally.

4 Introduction

1 The language mix

During the third ConTeXt conference that ran in parallel to EuroTeX 2009 in The Hague we had several sessions where MkIV was discussed and a few upcoming features were demonstrated. The next sections summarize some of that. It's hard to predict the future, especially because new possibilities show up once LuaTeX is opened up more, so remarks about the future are not definitive.

1.1 TeX

From now on, if I refer to TeX in the perspective of LuaTeX I mean "Good Old TeX", the language as well as the functionality. Although LuaTeX provides a couple of extensions it remains pretty close to compatible to its ancestor, certainly from the perspective of the end user.

As most ConTeXt users code their documents in the TeX language, this will remain the focus of MkIV. After all, there is no real reason to abandon it. However, although ConTeXt already stimulates users to use structure where possible and not to use low level TeX commands in the document source, we will add a few more structural variants. For instance, we already introduced `\startchapter` and `\startitem` in addition to `\chapter` and `\item`.

We even go further, by using key/value pairs for defining section titles, bookmarks, running headers, references, bookmarks and list entries at the start of a chapter. And, as we carry around much more information in the (for TeX so typical) auxiliary data files, we provide extensive control over rendering the numbers of these elements when they are recalled (like in tables of contents). So, if you really want to use different texts for all references to a chapter header, it can be done:

```
\startchapter
  [label=emcsquare,
   title={About $e=mc^2$},
   bookmark={einstein},
   list={About $e=mc^2$ (Einstein)},
   reference={$e=mc^2$}]

  ... content ...

\stopchapter
```

Under the hood, the MkIV code base is becoming quite a mix and once we have a more clear picture of where we're heading, it might become even more of a hybrid. Already for some time most of the font handling is done by Lua, and a bit more logic and management might move to Lua as well. However, as we want to be downward compatible we cannot go as far as we want (yet). This might change as soon as more of the primitives have associated Lua functions. Even then it will be a trade off: calling Lua takes some time and it might not pay off at all.

Some of the more tricky components, like vertical spacing, grid snapping, balancing columns, etc. are already in the process of being Luafied and their hybrid form might turn into complete Lua driven solutions eventually. Again, the compatibility issue forces us to follow a stepwise approach, but at the cost of (quite some) extra development time. But whatever happens, the TEX input language as well as machinery will be there.

1.2 MetaPost

I never regret integrating MetaPost support in ConTEXt and a dream came true when mplib became part of LuaTEX. Apart from a few minor changes in the way text integrates into MetaPost graphics the user interface in MkIV is the same as in MkII. Insofar as Lua is involved, this is hidden from the user. We use Lua for managing runs and conversion of the result to pdf. Currently generating MetaPost code by Lua is limited to assisting in the typesetting of chemical structure formulas which is now part of the core.

When defining graphics we use the MetaPost language and not some TEX-like variant of it. Information can be passed to MetaPost using special macros (like `\MPcolor`), but most relevant status information is passed automatically anyway.

You should not be surprised if at some point we can request information from TEX directly, because after all this information is accessible. Think of something `w := texdimen(0) ;` being expanded at the MetaPost end instead of `w := \the\dimen0 ;` being passed to MetaPost from the TEX end.

1.3 LUA

What will the user see of Lua? First of all he or she can use this scripting language to generate content. But when making a format or by looking at the statistics printed at the end of a run, it will be clear that Lua is used all over the place.

So how about LUA as a replacement for the TEX input language? Actually, it is already possible to make such "CONTEXT LUA Documents" using MKIV's built in functions. Each CONTEXT command is also available as a LUA function.

```
\startluacode
  context.bTABLE {
      framecolor = "blue",
      align= "middle",
      style = "type",
      offset=".5ex",
    }
    for i=1,10 do
      context.bTR()
      for i=1,20 do
        local r= math.random(99)
        if r < 50 then
          context.bTD {
            background = "color",
            backgroundcolor = "blue"
          }
          context(context.white("%#2i",r))
        else
          context.bTD()
          context("%#2i",r)
        end
        context.eTD()
      end
      context.eTR()
    end
  context.eTABLE()
\stopluacode
```

Of course it helps if you know CONTEXT a bit. For instance we can as well say:

```
if r < 50 then
  context.bTD {
    background = "color",
    backgroundcolor = "blue",
    foregroundcolor = "white",
  }
else
  context.bTD()
end
context("%#2i",r)
```

```
context.eTD()
```

And, knowing LUA helps as well, since the following is more efficient:

```
\startluacode
  local colored = {
    background = "color",
    backgroundcolor = "blue",
    foregroundcolor = "white",
  }
  local basespec = {
    framecolor = "blue",
    align= "middle",
    style = "type",
    offset=".5ex",
  }
  local bTR, eTR = context.bTR, context.eTR
  local bTD, eTD = context.bTD, context.eTD
  context.bTABLE(basespec)
    for i=1,10 do
      bTR()
      for i=1,20 do
        local r= math.random(99)
        bTD((r < 50 and colored) or nil)
        context("%#2i",r)
        eTD()
      end
      eTR()
    end
  context.eTABLE()
\stopluacode
```

Since in practice the speedup is negligible and the memory footprint is about the same, such optimization seldom make sense.

At some point this interface will be extended, for instance when we can use TEX's main (scanning, parsing and processing) loop as a so-called coroutine and when we have opened up more of TEX's internals. Of course, instead of putting this in your TEX source, you can as well keep the code at the LUA end.

The script that manages a CONTEXT run (also called `context`) will process files with the `cld` suffix automatically. You can also force processing as LUA with the flag `--forcecld`.[3] The `mtxrun` script also recognizes `cld` files and delegate

8 The language mix

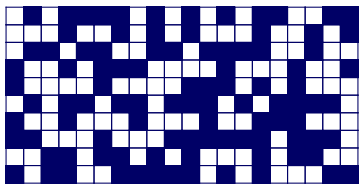


Figure 1.1 The result of the shown LUA code.

the call to the `context` script.

```
context yourfile.cld
```

But will this replace TEX as an input language? This is quite unlikely because coding documents in TEX is so convenient and there is not much to gain here. Of course in a pure LUA based workflow (for instance publishing information from databases) it would be nice to code in LUA, but even then it's mostly syntactic sugar, as TEX has to do the job anyway. However, eventually we will have a quite mature LUA counterpart.

1.4 XML

This is not so much a programming language but more a method of tagging your document content (or data). As structure is rather dominant in XML, it is quite handy for situations where we need different output formats and multiple tools need to process the same data. It's also a standard, although this does not mean that all documents you see are properly structured. This in turn means that we need some manipulative power in CONTEXT, and that happens to be easier to do in MKIV than in MKII.

In CONTEXT we have been supporting XML for a long time, and in MKIV we made the switch from stream based to tree based processing. The current implementation is mostly driven by what has been possible so far but as LUATEX becomes more mature, bits and pieces will be reimplemented (or at least cleaned up and brought up to date with developments in LUATEX).

One could argue that it makes more sense to use XSLT for converting XML into something TEX, but in most of the cases that I have to deal with much effort goes into mapping structure onto a given layout specification. Adding a bit of XML to TEX mapping to that directly is quite convenient. The total amount of code is probably smaller and it saves a processing step.

[3] Similar methods exist for processing XML files.

We're mostly dealing with education-related documents and these tend to have a more complex structure than the final typeset result shows. Also, readability of code is not served with such a split as most mappings look messy anyway (or evolve that way) due to the way the content is organized or elements get abused.

There is a dedicated manual for dealing with XML in MKIV, so we only show a simple example here. The documents to be processed are loaded in memory and serialized using setups that are associated to elements. We keep track of documents and nodes in a way that permits multipass data handling (rather usual in TEX). Say that we have a document that contains questions. The following definitions will flush the (root element) `questions`:

```
\startxmlsetups xml:mysetups
    \xmlsetsetup{#1}{questions}{xml:questions}
\stopxmlsetups

\xmlregistersetup{xml:mysetups}

\startxmlsetups xml:questions
    \xmlflush{#1}
\stopxmlsetups

\xmlprocessfile{main}{somefile.xml}{}
```

Here the `#1` represents the current XML element. Of course we need more associations in order to get something meaningful. If we just serialize then we have mappings like:

```
\xmlsetsetup{#1}{question|answer}{xml:*}
```

So, questions and answers are mapped onto their own setup which flushes them, probably with some numbering done at the spot.

In this mechanism LUA is sort of invisible but quite busy as it is responsible for loading, filtering, accessing and serializing the tree. In this case TEX and LUA hand over control in rapid succession.

You can hook in your own functions, like:

```
\xmlfilter{#1}{(wording|feedback|choice)/function(cleanup)}
```

In this case the function `cleanup` is applied to elements with names that match

one of the three given.[4]

Of course, once you start mixing in LUA in this way, you need to know how we deal with XML at the LUA end. The following function show how we calculate scores:

```
\startluacode
function xml.functions.totalscore(root)
  local n = 0
  for e in xml.collected(root,"/outcome") do
    if xml.filter(e,"action[text()='add']") then
      local m = xml.filter(e,"xml:///score/text()")
      n = n + (tonumber(m or 0) or 0)
    end
  end
  tex.write(n)
end
\stopluacode
```

You can either use such a function in a filter or just use it as a TEX macro:

```
\startxmlsetups xml:question
  \blank
  \xmlfirst{#1}{wording}
  \startitemize
    \xmlfilter{#1}{/answer/choice/command(xml:answer:choice)}
  \stopitemize
  \endgraf
  score: \xmlfunction{#1}{totalscore}
  \blank
\stopxmlsetups

\startxmlsetups xml:answer:choice
    \startitem
        \xmlflush{#1}
    \stopitem
\stopxmlsetups
```

The filter variant is like this:

[4] This example is inspired by one of our projects where the cleanup involves sanitizing (highly invalid) HTML data that is embedded as a `CDATA` stream, a trick to prevent the XML file to be invalid.

```
\xmlfilter{#1}{./function('totalscore')}
```

So you can take your choice and make your source look more XML-ish, LUA-like or TEX-wise. A careful reader might have noticed the peculiar `xml://` in the function code. When used inside MKIV, the serializer defaults to TEX so results are piped back into TEX. This prefix forced the regular serializer which keeps the result at the LUA end.

Currently some of the XML related modules, like MATHML and handling of tables, are really a mix of TEX code and LUA calls, but it makes sense to move them completely to LUA. One reason is that their input (formulas and table content) is restricted to non-TEX anyway. On the other hand, in order to be able to share the implementation with TEX input, it also makes sense to stick to some hybrid approach. In any case, more of the calculations and logic will move to LUA, while TEX will deal with the content.

A somewhat strange animal here is XSL-FO. We do support it, but the MKII implementation was always somewhat limited and the code was quite complex. So, this needs a proper rewrite in MKIV, which will happen indeed. It's mostly a nice exercise of hybrid technology but until now I never really needed it. Other bits and pieces of the current XML goodies might also get an upgrade.

There is already a bunch of functions and macros to filter and manipulate XML content and currently the code involved is being cleaned up. What direction we go also depends on users' demands. So, with respect to XML you can expect more support, a better integration and an upgrade of some supported XML related standards.

1.5 Tools

Some of the tools that ship with CONTEXT are also examples of hybrid usage.

Take this:

```
mtxrun --script server --auto
```

On my machine this reports:

```
MTXrun | running at port: 31415
MTXrun | document root: c:/data/develop/context/lua
MTXrun | main index file: unknown
MTXrun | scripts subpath: c:/data/develop/context/lua
MTXrun | context services: http://localhost:31415/mtx-server-ctx-startup.lua
```

The `mtxrun` script is a LUA script that acts as a controller for other scripts, in this case `mtx-server.lua` that is part of the regular distribution. As we use LUATEX as a LUA interpreter and since LUATEX has a socket library built in, it can act as a web server, limited but quite right for our purpose.[5]

The web page that pops up when you enter the given address lets you currently choose between the CONTEXT help system and a font testing tool. In figure 1.2 you seen an example of what the font testing tool does.

Figure 1.2 An example of using the font tester.

Here we have LUATEX running a simple web server but it's not aware of having TEX on board. When you click on one of the buttons at the bottom of the screen, the server will load and execute a script related to the request and in this case that script will create a TEX file and call LUATEX with CONTEXT to process that file. The result is piped back to the browser.

You can use this tool to investigate fonts (their bad and good habits) as well as to test the currently available OPENTYPE functionality in MKIV (bugs as well as goodies).

So again we have a hybrid usage although in this case the user is not confronted with LUA and/or TEX at all. The same is true for the other goodie, shown in figure 1.3. Actually, such a goodie has always been part of the CONTEXT

[5] This application is not intentional but just a side effect.

distribution but it has been rewritten in LUA.

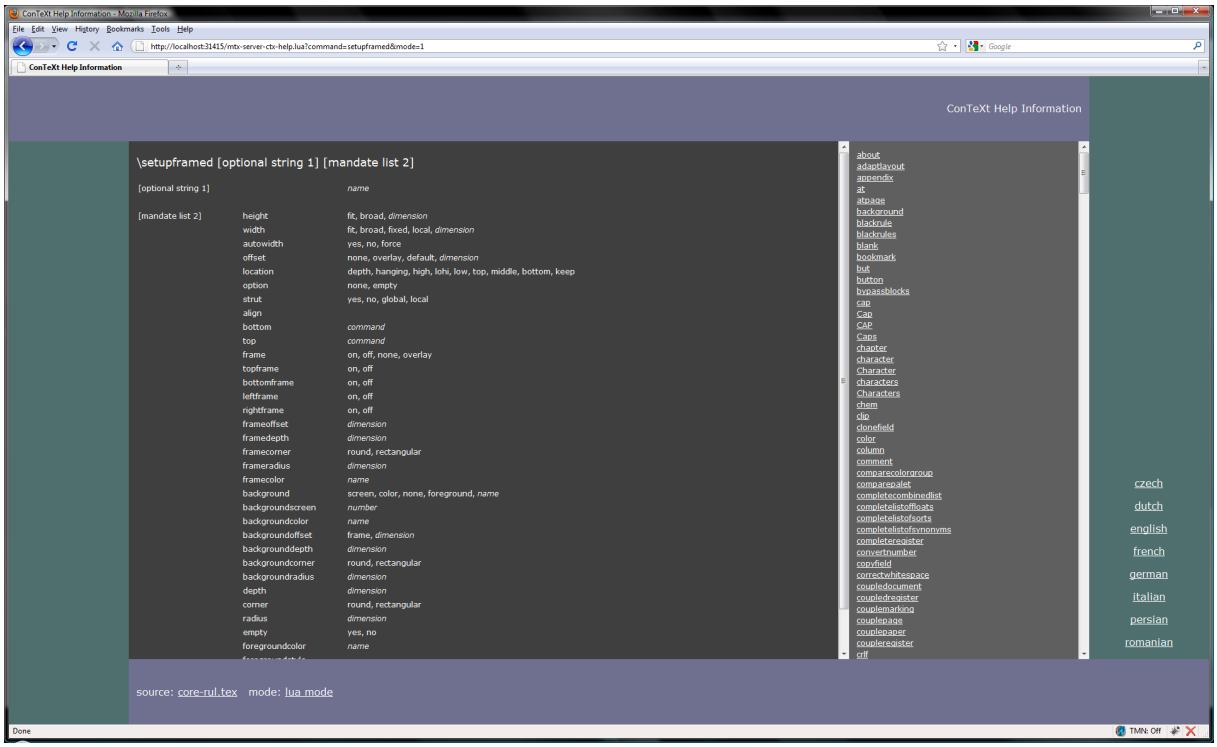


Figure 1.3 An example of a help screen for a command.

The CONTEXT user interface is defined in an XML file, and this file is used for several purposes: initializing the user interfaces at format generation time, typesetting the formal command references (for all relevant interface languages), for the wiki, and for the mentioned help goodie.

Using the mix of languages permits us to provide convenient processing of documents that otherwise would demand more from the user than it does now. For instance, imagine that we want to process a series of documents in the so-called Epub format. Such a document is a zipped file that has a description and resources. As the content of this archive is prescribed it's quite easy to process it:

```
context --ctx=x-epub.ctx yourfile.epub
```

This is equivalent to:

```
texlua mtxrun.lua --script context --ctx=x-epub.ctx yourfile.epub
```

So, here we have LUATEX running a script that itself (locates and) runs a script `context`. That script loads a CONTEXT job description file (with suffix `ctx`). This file tells what styles to load and might have additional directives but none of that has to bother the end user. In the automatically loaded style we take care

of reading the XML files from the zipped file and eventually map the embedded HTML like files onto style elements and produce a PDF file. So, we have LUA managing a run and MKIV managing with help of LUA reading from zip files and converting XML into something that TEX is happy with. As there is no standard with respect to the content itself, i.e. the rendering is driven by whatever kind of structure is used and whatever the CSS file is able to map it onto, in practice we need an additional style for this class of documents. But anyway it's a good example of integration.

1.6 The future

Apart from these language related issues, what more is on the agenda? To mention a few integration related thoughts:

- At some point I want to explore the possibility to limit processing to just one run, for instance by doing trial runs without outputting anything but still collecting multipass information. This might save some runtime in demanding workflows especially when we keep extensive font loading and image handling in mind.
- Related to this is the ability to run MKIV as a service but that demands that we can reset the state of LUATEX and actually it might not be worth the trouble at all given faster processors and disks. Also, it might not save much runtime on larger jobs.
- More interesting can be to continue experimenting with isolating parts of CONTEXT in such a way that one can construct a specialized subset of functionality. Of course the main body of code will always be loaded as one needs basic typesetting anyway.

Of course we keep improving existing mechanisms and improve solutions using a mix of TEX and LUA, using each language (and system) for what it can do best.

16 The language mix

2 Font Goodies

2.1 Introduction

The Oriental TEX project is one of the first and more ambitious users of LuaTEX. A major undertaking in this project is the making of a rather full features and complex font for typesetting Arabic. As the following text will show some Arabic, you might get the impression that I'm an expert but be warned that I'm far from that. But as Idris compensates this quite well the team has a lot of fun in figuring out how to achieve our goals using OpenType technology in combination with LuaTEX and MkIV. A nice side effect of this is that we end up with some neat tricks in the ConTEXt core.

Before we come to some of these goodies, an example of Arabic is given that relates quite well to the project. It was first used at the euroTEX 2009 meeting. Take the following 6 shapes:

With these we can make the name LuaTEX and as we use a nice script we can forget about the lowered E. Putting these characters in sequence is not enough as Arabic typesetting has to mimick the subtle aspects of scribes.

In Latin scripts we have mostly one-to-one and many-to-one substitutions. These can happen in sequence which in in practice boils down to multiple passes over the stream of characters. In this process sometimes surrounding characters (or shapes) play a role, for instance ligatures are not always wanted and their coming into existence might depend on neighbouring characters. In some cases glyphs have to be (re)positioned relative to each other. While in Latin scripts the number of substitutions and positioning is not that large but in advanced Arabic fonts it can be pretty extensive.

With OpenType we have some machinery available, so we try to put as much logic in the font as possible. However, in addition we have some dedicated optimizing routines. The whole process is split into a couple if stages.

The so called First-Order Analysis puts a given character into isolated, initial, middle, or final state. Next, the Second-Order Analysis looks at the characters and relates this state to what characters precede or succeed it. Based on that state we do character substitutions. There can be multiple analysis and

replacements in sequence. We can do some simple aesthetic stretching and additional related replacements. We need to attach identity marks and vowels in proper but nice looking places. In most cases we're then done. Contrary to other fonts we don't use many ligatures but compose characters.

The previous steps already give reasonable results and implementing it also nicely went along with the development of LuaTEX and ConTEXt MkIV. Currently we're working on extending and perfecting the font to support what we call Third-Order Contextual Analysis. This boils down to an interplay between the paragraph builder and additional font features. In order to get pleasing spacing we apply further substitutions, this time with wider or narrower shapes. When this is done we need to reattach identity marks and vowels. Optionally we can apply hz like stretching as a finishing touch but so far we didn't follow that route yet.

So, let's see how we can typeset the word LuaTEX in Arabic using some of these techniques.

no order (kh ī t ā w [u] l) لـُواتيخ

first order لُواتيخ

second order لُواتيخ

second order (Jiim-stacking) لُواتيخ

minimal stretching لُواتيخ

maximal stretching (level 3) لُـواتـيخ

chopped letter khaa (for e.g. underlining) 

As said, this font is quite complex in the sense that it has many features and associated lookups. In addition to the usual features we have stylistic and justification variants. As these are not standardized (after all, each font can have its own look and feel and associated treatments) we store some information in the goodies files that ship with this font.

feature	meaning
js01	Raawide
js02	Yaawide
js03	Kaafwide
js04	Nuunwide
js05	Kaafwide Nuunwide Siinwide Baawide
js06	final Haa wide
js07	thin Miim
js08	short Miim
js09	wide Siin
js10	thuluth-style initial Haa, final Miim, MRw_mf
js11	level-1 stretching
js12	level-2 stretching
js13	level-3 stretching
js14	final Alif
js15	hooked final Alif
js16	aesthetic medial Faa/Qaaf
js17	fancy isol Haa after Daal, Raa, and Waaw
js18	Laamwide, alternate substitution
js19	level-4 stretching, only siin and Hhaa for basmalah
js20	level-5 stretching, only siin and Hhaa for basmalah
js21	Haa.final_alt2
ss01	Allah, Muhammad
ss02	ss01 + Allah_final
ss03	level-1 stack over Jiim, initial entry only
ss04	level-1 stack over Jiim, initial/medial entry
ss05	multi-level Jiim stacking, initial/medial entry
ss06	aesthetic Faa/Qaaf for FJ_mm, FJ_mf connection
ss07	initial-entry stacking over Haa
ss08	initial/medial stacking over Haa, minus HM_mf strings
ss09	initial/medial Haa stacking plus HM_mf strings
ss10	basic dipped Miim, initial-entry B_S-stack over Miim
ss11	full dipped Miim, initial-entry B_S-stack over Miim

```
ss12    XBM_im initial-medial entry B_S-stack over Miim
ss13    full initial-medial entry B_S-stacked Miim
ss14    initial entry, stacked Laam on Miim
ss15    full stacked Laam-on-Miim
ss16    initial entry, stacked Ayn-on-Miim
ss17    full stacked Ayn-on-Miim
ss18    LMJ_im already contained in ss03--05, may remove
ss19    LM_im
ss20    KLM_m, sloped Miim
ss21    KLM_i_mm/LM_mm, sloped Miim
ss22    filled sloped Miim
ss23    LM_mm, non-sloped Miim
ss24    BR_i_mf, BN_i_mf
ss25    basic LH_im might merge with ss24
ss26    full Yaa.final special strings: BY_if, BY_mf, LY_mf
ss27    basic thin Miim.final
ss28    full thin Miim.final to be moved to jsnn
ss29    basic short Miim.final
ss30    full short Miim.final to be moved to jsnn
ss31    basic Raa.final strings: JR and SR
ss32    basic Raa.final strings: JR, SR, and BR
ss33    TtR to be moved to jsnn
ss34    AyR style also available in jsnn
ss35    full Kaaf contexts
ss36    full Laam contexts
ss37    Miim-Miim contexts
ss38    basic dipped Haa, B_SH_mm
ss39    full dipped Haa, B_S_LH_i_mm_Mf
ss40    aesthetic dipped medial Haa
ss41    high and low Baa strings
ss42    diagonal entry
ss43    initial alternates
ss44    hooked final alif
ss45    BMA_f
ss46    BM_mm_alt, for JBM combinations
ss47    Shaddah-<kasrah> combo
ss48    Auto-sukuun
ss49    No vowels
ss50    Shaddah/MaaddahHamzah only
ss51    No Skuun
ss52    No Waslah
ss53    No Waslah
ss54    chopped finals
ss55    idgham-tanwin
```

20 Font Goodies

It is highly unlikely that a user will remember all these features, which is why there will be a bunch of predefined combinations. These are internalized as follows:

featureset	definitions
default	analyze=true anum=true calt=true ccmp=true curs=true fina=true init=true js16=true kern=true language=dflt mark=true medi=true mkmk=true mode=node number=108 rlig=true salt=true script=arab ss01=true ss03=true ss07=true ss10=true ss12=true ss15=true ss16=true ss19=true ss24=true ss25=true ss26=true ss27=true ss31=true ss34=true ss35=true ss36=true ss37=true ss38=true ss41=true ss42=true ss43=true ss55=true
maximal_stretching	analyze=true anum=true calt=true ccmp=true curs=true fina=true init=true js05=true js09=true js13=true js16=true kern=true language=dflt mark=true medi=true mkmk=true mode=node number=112 rlig=true salt=true script=arab ss01=true ss03=true ss07=true ss10=true ss12=true ss15=true ss16=true ss19=true ss24=true ss25=true ss26=true ss27=true ss31=true ss34=true ss35=true ss36=true ss37=true ss38=true ss41=true ss42=true ss43=true ss55=true
medium_stretching	analyze=true anum=true calt=true ccmp=true curs=true fina=true init=true js05=true js12=true js16=true kern=true language=dflt mark=true medi=true mkmk=true mode=node number=113 rlig=true salt=true script=arab ss01=true ss03=true ss07=true ss10=true ss12=true ss15=true ss16=true ss19=true ss24=true ss25=true ss26=true ss27=true ss31=true ss34=true ss35=true ss36=true ss37=true ss38=true ss41=true ss42=true ss43=true ss55=true
minimal_stretching	analyze=true anum=true calt=true ccmp=true curs=true fina=true init=true js03=true js11=true js16=true kern=true language=dflt mark=true medi=true mkmk=true mode=node number=110 rlig=true salt=true script=arab ss01=true ss03=true ss07=true ss10=true ss12=true ss15=true ss16=true ss19=true

```
                    ss24=true  ss25=true  ss26=true  ss27=true
                    ss31=true  ss34=true  ss35=true  ss36=true
                    ss37=true  ss38=true  ss41=true  ss42=true
                    ss43=true  ss55=true
shrink              analyze=true  anum=true  calt=true  ccmp=true
                    curs=true  fina=true  flts=true  init=true
                    js16=true  js17=true  kern=true  language=dflt
                    mark=true  medi=true  mkmk=true  mode=node
                    number=111  rlig=true  salt=true  script=arab
                    ss01=true  ss03=true  ss05=true  ss06=true
                    ss07=true  ss09=true  ss10=true  ss11=true
                    ss12=true  ss15=true  ss16=true  ss19=true
                    ss24=true  ss25=true  ss26=true  ss27=true
                    ss31=true  ss34=true  ss35=true  ss36=true
                    ss37=true  ss38=true  ss41=true  ss42=true
                    ss43=true  ss55=true
wide_all            analyze=true  anum=true  calt=true  ccmp=true
                    curs=true  fina=true  init=true  js05=true
                    js09=true  js11=true  js12=true  js13=true
                    js16=true  kern=true  language=dflt  mark=true
                    medi=true  mkmk=true  mode=node  number=109
                    rlig=true  salt=true  script=arab  ss01=true
                    ss03=true  ss07=true  ss10=true  ss12=true
                    ss15=true  ss16=true  ss19=true  ss24=true
                    ss25=true  ss26=true  ss27=true  ss31=true
                    ss34=true  ss35=true  ss36=true  ss37=true
                    ss38=true  ss41=true  ss42=true  ss43=true
                    ss55=true
```

2.2 Color

One of the objectives of the oriental TEX project is to bring color to typeset Arabic. When Idris started making samples with much manual intervention it was about time to figure out if it could be supported by a bit of LUA code.

As the colorization concerns classes of glyphs (like vowels) this is something that can best be done after all esthetics have been sorted out. Because things like coloring are not part of font technology and because we don't want to misuse the OPENTYPE feature mechanisms for that, the solution lays in an extra file that describes these goodies.

لُواتيخ أألف ليلة وليلة

لُواتيخ أألف ليلة وليلة

لُواتيخ أألف ليلة وليلة

The second and third of these three lines have colored vowels and identity marks. So how did we get the colors? There are actually two mechanisms involved in this:

- we need to associate colorschemes with classed of glyphs
- we need to be able to turn on and off coloring

The first is done by loading goodies and selecting a colorscheme:

```
\definefontfeature
  [husayni-colored]
  [goodies=husayni,
   colorscheme=default,
   featureset=default]
```

Turning on and off coloring is done with two commands (we might provide a proper environment for this) as shown in:

```
\start
  \definedfont[husayni*husayni-colored at 72pt]
  \righttoleft
  \resetfontcolorscheme   لُواتيخ أألف ليلة وليلة \par
  \setfontcolorscheme  [1]لُواتيخ أألف ليلة وليلة \crlf
```

```
  \setfontcolorscheme  [2]لُواتيخ ألف ليلة وليلة \crlf
\stop
```

If you look closely at the feature definition you'll notice that we also choose a default featureset. For most (latin) fonts the regular feature definitions are convenient, but for fonts that are used for Arabic there are preferred combinations of features as there can be many.

Currently the font we use here has the following colorschemes:

colorscheme	numbers
default	1 2 3 4 5

2.3 The goodies file

In principle a goodies files can contain anuy data that makes sense but in order to be useable some entries have a prescribed structure. A goodies file looks as follows:

```
return {
  name = "husayni",
  version = "1.00",
  comment = "Goodies that complement the Husayni font by Idris Samawi Hamid.",
  author = "Idris Samawi Hamid and Hans Hagen",
  featuresets = {
    default = {
      key = value, <table>, ...
    },
    ...
  },
  stylistics = {
    key = value, ...
  },
  colorschemes = {
    default = {
      [1] = {
        "glyph_a.one", "glyph_b.one", ...
      },
      ...
    }
  }
}
```

We already saw the list of special features and these are defined in the `stylistics` stable. In this document, that list was typeset using the following (hybrid) code:

```
\startluacode
  local goodies = fonts.goodies.load("husayni")
  local stylistics = goodies and goodies.stylistics
  if stylistics then
    local col, row, type = context.NC, context.NR, context.type
    context.starttabulate { "|l|pl|" }
    col() context("feature") col() context("meaning") col() row()
    for feature, meaning in table.sortedpairs(stylistics) do
      col() type(feature) col() type(meaning) col() row()
    end
    context.stoptabulate()
  end
\stopluacode
```

The table with colorscheme that we showed is generated with:

```
colorscheme   numbers
default       1  2  3  4  5
```

In a similar fashion we typeset the featuresets:

```
\startluacode
  local goodies = fonts.goodies.load("husayni")
  local featuresets = goodies and goodies.featuresets
  if featuresets then
    local col, row, type = context.NC, context.NR, context.type
    context.starttabulate { "|l|pl|" }
    col() context("featureset") col() context("definitions") col() row()
    for featureset, definitions in table.sortedpairs(featuresets) do
      col() type(featureset) col()
      for k, v in table.sortedpairs(definitions) do
        type(string.format("%s=%s",k,tostring(v)))
        context.quad()
      end
      col() row()
    end
    context.stoptabulate()
  end
\stopluacode
```

The unprocessed `featuresets` table can contain one or more named sets and

each set can be a mixture of tables and key value pairs. Say that we have:

```
default = {
  kern = "yes", { ss01 = "yes" }, { ss02 = "yes" }, "mark"
}
```

Given the previous definition, the order of processing is as follows.

1. `{ ss01 = "yes" }`
2. `{ ss02 = "yes" }`
3. `mark` (set to `"yes"`)
4. `kern = "yes"`

So, first we process the indexed part if the list, and next the hash. Already set values are not set again. The advantage of using a LUA table is that you can simplify definitions. Before we return the table we can define local variables, like:

```
local one = { ss01 = "yes" }
local two = { ss02 = "yes" }
local pos = { kern = "yes", mark = "yes" }
```

and use them in:

```
default = {
  one, two, pos
}
```

That way we we can conveniently define all kind of interesting combinations without the need for many repetitive entries.

The `colorsets` table has named subtables that are (currently) indexed by number. Each number is associated with a color (at the TEX end) and is coupled to a list of glyphs. As you can see here, we use the name of the glyph. We prefer this over an index (that can change during development of the font). We cannot use UNICODE points as many such glyphs are just variants and have no unique code.

2.4 Optimizing Arabic

The ultimate goal of the Oriental TEX project is to improve the look and feel of a paragraph. Because TEX does a pretty good job on breaking the paragraph

into lines, and because complicating the paragraph builder is not a good idea, we finally settled on improving the lines that result from the par builder. This approach is rather close to what scribes do and the advanced Husayni font provides features that support this.

In principle the current optimizer can replace character expansion but that would slow down considerably. Also, for that we first have to clean up the experimental LUA based par builder.

After several iterations the following approach was chosen.

- We typeset the paragraph with an optimal feature set. In our case this is `husayni-default`.

- Next we define two sets of additional features: one that we can apply to shrink words, and one that does the opposite.

- When the line has a badness we don't like, we either stepwise shrink words or stretch them, depending on how bad things are.

The set that takes care of shrinking is defined as:

```
\definefontfeature
  [shrink]
  [husayni-default]
  [flts=yes,js17=yes,ss05=yes,ss11=yes,ss06=yes,ss09=yes]
```

Stretch has a few more variants:

```
\definefontfeature
  [minimal_stretching]
  [husayni-default]
  [js11=yes,js03=yes]
\definefontfeature
  [medium_stretching]
  [husayni-default]
  [js12=yes,js05=yes]
\definefontfeature
  [maximal_stretching]
  [husayni-default]
  [js13=yes,js05=yes,js09=yes]
\definefontfeature
  [wide_all]
  [husayni-default]
```

```
  [js11=yes,js12=yes,js13=yes,js05=yes,js09=yes]
```

Next we define a font solution:

```
\definefontsolution
  [FancyHusayni]
  [goodies=husayni,
   less=shrink,
   more={minimal_stretching,medium_stretching,maximal_stretching,wide_all}]
```

Because these featuresets relate quite closely to the font design we don't use this way if defining but put the definitions in the goodies file:

```
.....
featuresets = { -- here we don't have references to featuresets
    default = {
        default,
    },
    minimal_stretching = {
        default, js11 = yes, js03 = yes,
    },
    medium_stretching = {
        default, js12=yes, js05=yes,
    },
    maximal_stretching= {
        default, js13 = yes, js05 = yes, js09 = yes,
    },
    wide_all = {
        default, js11 = yes, js12 = yes, js13 = yes, js05 = yes, js09 = yes,
    },
    shrink = {
        default, flts = yes, js17 = yes, ss05 = yes, ss11 = yes, ss06 = yes, ss09 = yes,
    },
},
solutions = { -- here we have references to featuresets, so we use strings!
    experimental = {
        less = { "shrink" },
        more = { "minimal_stretching", "medium_stretching", "maximal_stretching", "wide_all" },
    },
},
.....
```

Now the definition looks much simpler:

```
\definefontsolution
  [FancyHusayni]
  [goodies=husayni,
```

```
    solution=experimental]
```

I want some funny text (complete with translation). Actually I want all examples translated.

In the following example the yellow words are stretched and the green ones are shrunken.[6]

```
\definedfont[husayni*husayni-default at 24pt]
% todo: factor ivm grid, so the next line looks hackery:
\expanded{\setuplocalinterlinespace[line=\the\dimexpr2\lineheight]}
\setfontsolution[FancyHusayni]% command will change
\enabletrackers[builders.paragraphs.solutions.splitters.colors]
\righttoleft \getbuffer[sample] \par
\disabletrackers[builders.paragraphs.solutions.splitters.colors]
\resetfontsolution
```

قد صعدنا ذرى الحقائق بأقدام النبوة و الولاية و نورنا سبع طبقات أعلام الفتوى بالهداية
فنحن ليوث الوغى و غيوث الندى و طعان العدى و فينا السيف و القلم في العاجل و
لواء الحمد و الحوض في الآجل و أسباطنا حلفاء الدين و خلفاء النبيين و مصابيح الأمم و
مفاتيح الكرم فالكليم ألبس حلة الاصطفاء لما عهدنا منه الوفاء و روح القدس في جنان
الصاقورة ذاق من حدائقنا الباكورة و شيعتنا الفئة الناجية و الفرقة الزاكية و صاروا لنا
ردءا و صونا و على الظلمة ألبا و عونا و سينفجر لهم ينابيع الحيوان بعد لظى النيران لثمام
آل حم و طه و الطواسين من السنين و هذا الكتاب درة من درر الرحمة و قطرة من بحر
الحكمة و كتب الحسن بن علي العسكري في سنة أربع و خمسين و مائتين

This mechanism is somewhat experimental as is the (user) interface. It is also rather slow compared to normal processing. There is room for improvement but I will do that when other components are more stable so that simple variants (that we can use here) can be derived.

When criterium 0 used above is changed into for instance 5 processing is faster. When you enable a preroll processing is more time consuming. Examples of

[6] Make sure that the paragraph is finished (for instance using `\par` before resetting it.)

settings are:

```
\setupfontsolutions[method={preroll,normal},criterium=2]
\setupfontsolutions[method={preroll,random},criterium=5]
\setupfontsolutions[method=reverse,criterium=8]
\setupfontsolutions[method=random,criterium=2]
```

Using a preroll is slower because it first tries all variants and then settles for the best; otherwise we process the first till the last solution till the criterium is satisfied.

2.5 Protrusion and expansion

There are two entries in the goodies file that relate to advanced parbuilding: `protrusions` and `expansions`.

```
protrusions = {
  vectors = {
    pure = {
      [0x002C] = { 0, 1 }, -- comma
      [0x002E] = { 0, 1 }, -- period
      .....
    }
  }
}
```

These vectors are similar to the ones defined globally but the vectors defined in a goodie file are taken instead when present.

2.6 Filenames and properties

As filenames and properties of fonts are somewhat of an inconsistent mess, we can use the goodies to provide more information:

```
files = {
  name = "antykwapoltawskiego", -- shared
  list = {
      ["AntPoltLtCond-Regular.otf"] = {
       -- name   = "antykwapoltawskiego",
          style  = "regular",
          weight = "light",
```

```
          width  = "condensed",
      },
      .....
    }
  }
}
```

Internally this will become a lookup tree so that we can have a predictable specifier:

```
\definefont[MyFontA][antykwapoltawskiego-bold-italic]
\definefont[MyFontB][antykwapoltawskiego-normal-italic-condensed]
\definefont[MyFontC][antykwapoltawskiego-light-regular-semicondensed]
```

Of course one needs to load the goodies. One way to force that is:

```
\loadfontgoodies[antykwapoltawskiego]
```

The Antykwa Poltawskiego family is rather large and provides all kind of combinations.

antykwapoltawskiego-bold-regular-normal-normal
antykwapoltawskiego-bold-italic-normal-normal
antykwapoltawskiego-normal-italic-normal-normal
antykwapoltawskiego-normal-regular-normal-normal
antykwapoltawskiego-bold-regular-condensed-normal
antykwapoltawskiego-bold-italic-condensed-normal
antykwapoltawskiego-normal-italic-condensed-normal
antykwapoltawskiego-normal-regular-condensed-normal
antykwapoltawskiego-bold-regular-expanded-normal
antykwapoltawskiego-bold-italic-expanded-normal
antykwapoltawskiego-normal-italic-expanded-normal
antykwapoltawskiego-normal-regular-expanded-normal
antykwapoltawskiego-medium-regular-normal-normal
antykwapoltawskiego-medium-italic-normal-normal
antykwapoltawskiego-light-italic-normal-normal
antykwapoltawskiego-light-regular-normal-normal
antykwapoltawskiego-medium-regular-condensed-normal
antykwapoltawskiego-medium-italic-condensed-normal
antykwapoltawskiego-light-italic-condensed-normal
antykwapoltawskiego-light-regular-condensed-normal
antykwapoltawskiego-medium-regular-expanded-normal
antykwapoltawskiego-medium-italic-expanded-normal

antykwapoltawskiego-light-italic-expanded-normal
antykwapoltawskiego-light-regular-expanded-normal
antykwapoltawskiego-medium-regular-semicondensed-normal
antykwapoltawskiego-medium-italic-semicondensed-normal
antykwapoltawskiego-light-italic-semicondensed-normal
antykwapoltawskiego-light-regular-semicondensed-normal
antykwapoltawskiego-medium-regular-semiexpanded-normal
antykwapoltawskiego-medium-italic-semiexpanded-normal
antykwapoltawskiego-light-italic-semiexpanded-normal
antykwapoltawskiego-light-regular-semiexpanded-normal
antykwapoltawskiego-bold-regular-semicondensed-normal
antykwapoltawskiego-bold-italic-semicondensed-normal
antykwapoltawskiego-normal-italic-semicondensed-normal
antykwapoltawskiego-normal-regular-semicondensed-normal
antykwapoltawskiego-bold-regular-semiexpanded-normal
antykwapoltawskiego-bold-italic-semiexpanded-normal
antykwapoltawskiego-normal-italic-semiexpanded-normal
antykwapoltawskiego-normal-regular-semiexpanded-normal

This list is generated with:

```
\usemodule[fonts-goodies]
\showfontgoodiesfiles[name=antykwapoltawskiego]
```

3 Grouping

3.1 Variants

After using TEX for a while you get accustomed to one of its interesting concepts: grouping. Programming languages like PASCAL and MODULA have keywords `begin` and `end`. So, one can say:

```
if test then begin
    print_bold("test 1")
    print_bold("test 2")
end
```

Other languages provide a syntax like:

```
if test {
    print_bold("test 1")
    print_bold("test 2")
}
```

So, in those languages the `begin` and `end` and/or the curly braces define a 'group' of statements. In TEX on the other hand we have:

```
test \begingroup \bf test \endgroup test
```

Here the second `test` comes out in a bold font and the switch to bold (basically a different font is selected) is reverted after the group is closed. So, in TEX grouping deals with scope and not with grouping things together.

In other languages it depends on the language of locally defined variables are visible afterwards but in TEX they're really local unless a `\global` prefix (or one of the shortcuts) is used.

In languages like LUA we have constructs like:

```
for i=1,100 do
    local j = i + 20
    ...
end
```

Here `j` is visible after the loop ends unless prefixed by `local`. Yet another example is MetaPost:

```
begingroup ;
    save n ; numeric n ; n := 10 ;
    ...
endgroup ;
```

Here all variables are global unless they are explicitly saved inside a group. This makes perfect sense as the resulting graphic also has a global (accumulated) property. In practice one rarely needs grouping, contrary to TEX where one really wants to keep changes local, if only because document content is so unpredictable that one never knows when some change in state happens.

In principle it is possible to carry over information across a group boundary. Consider this somewhat unrealistic example:

```
\begingroup
    \leftskip 10pt
    \begingroup
        ....
        \advance\leftskip 10pt
        ....
    \endgroup
\endgroup
```

How do we carry the advanced leftskip over the group boundary without using a global assignment which could have more drastic side effects? Here is the trick:

```
\begingroup
    \leftskip 10pt
    \begingroup
        ....
        \advance\leftskip 10pt
        ....
        \expandafter
    \endgroup
    \expandafter \leftskip \the\leftskip
\endgroup
```

This is typical the kind of code that gives new users the creeps but normally they never have to do that kind of coding. Also, that kind of tricks assumes that one knows how many groups are involved.

3.2 Implication

What does this all have to do with LuaTEX and MkIV? The user interface of ConTEXt provide lots of commands like:

```
\setupthis[style=bold]
\setupthat[color=green]
```

Most of them obey grouping. However, consider a situation where we use Lua code to deal with some aspect of typesetting, for instance numbering lines or adding ornamental elements to the text. In ConTEXt we flag such actions with attributes and often the real action takes place a bit later, for instance when a paragraph or page becomes available.

A comparable pure TEX example is the following:

```
{test test \bf test \leftskip10pt test}
```

Here the switch to bold happens as expected but no leftskip of 10pt is applied. This is because the set value is already forgotten when the paragraph is actually typeset. So in fact we'd need:

```
{test test \bf test \leftskip10pt test \par}
```

Now, say that we have:

```
{test test test \setupflag[option=1] \flagnexttext test}
```

We flag some text (using an attribute) and expect it to get a treatment where option 1 is used. However, the real action might take place when TEX deals with the paragraph or page and by that time the specific option is already forgotten or it might have gotten another value. So, the rather natural TEX grouping does not work out that well in a hybrid situation.

As the user interface assumes a consistent behaviour we cannot simply make these settings global even if this makes much sense in practice. One solution is to carry the information with the flagged text i.e. associate it somehow in the attribute's value. Of course, as we never know in advance when this information is used, this might result in quite some states being stored persistently.

A side effect of this 'problem' is that new commands might get suboptimal user interfaces (especially inheritance or cloning of constructs) that are somewhat driven by these 'limitations'. Of course we may wonder if the end user will notice this.

To summarize this far, we have three sorts of grouping to deal with:

- TEX's normal grouping model limits its scope to the local situation and normally has only direct and local consequences. We cannot carry information over groups.

- Some of TEX's properties are applied later, for instance when a paragraph or page is typeset and in order to make 'local' changes effective, the user needs to add explicit paragraph ending commands (like `\par` or `\page`).

- Features dealt with asynchronously by LUA are at that time unaware of grouping and variables set that were active at the time the feature was triggered so there we need to make sure that our settings travel with the feature. There is not much that a user can do about it as this kind of management has to be done by the feature itself.

It is the third case that we will give an example of in the next section. We leave it up to the user if it gets noticed on the user interface.

3.3 An example

A group of commands that has been reimplemented using a hybrid solution is underlining or more generic: bars. Just take a look at the following examples and try to get an idea on how to deal with grouping. Keep in mind that:

- Colors are attributes and are resolved in the backend, so way after the paragraph has been typeset.
- Overstrike is also handled by an attribute and gets applied in the backend as well, before colors are applied.
- Nested overstrikes might have different settings.
- An overstrike rule either inherits from the text or has its own color setting.

First an example where we inherit color from the text:

```
\definecolor[myblue][b=.75]
\definebar[myoverstrike][overstrike][color=]

Test \myoverstrike{%
    Test \myoverstrike{\myblue
        Test \myoverstrike{Test}
        Test}
    Test}
Test
```

Test ~~Test~~ ~~Test~~ ~~Test~~ ~~Test~~ ~~Test~~ Test

Because color is also implemented using attributes and processed later on we can access that information when we deal with the bar.

The following example has its own color setting:

```
\definecolor[myblue][b=.75]
\definecolor[myred] [r=.75]
\definebar[myoverstrike][overstrike][color=myred]

Test \myoverstrike{%
    Test \myoverstrike{\myblue
        Test \myoverstrike{Test}
        Test}
    Test}
Test
```

Test ~~Test~~ ~~Test~~ ~~Test~~ ~~Test~~ ~~Test~~ Test

See how we can color the levels differently:

```
\definecolor[myblue] [b=.75]
\definecolor[myred]  [r=.75]
\definecolor[mygreen][g=.75]

\definebar[myoverstrike:1][overstrike][color=myblue]
\definebar[myoverstrike:2][overstrike][color=myred]
\definebar[myoverstrike:3][overstrike][color=mygreen]

Test \myoverstrike{%
    Test \myoverstrike{%
        Test \myoverstrike{Test}
        Test}
    Test}
Test
```

Test ~~Test~~ ~~Test~~ ~~Test~~ ~~Test~~ ~~Test~~ Test

Watch this:

```
\definecolor[myblue] [b=.75]
\definecolor[myred]  [r=.75]
\definecolor[mygreen][g=.75]
```

```
\definebar[myoverstrike][overstrike][max=1,dy=0,offset=.5]
\definebar[myoverstrike:1][myoverstrike][color=myblue]
\definebar[myoverstrike:2][myoverstrike][color=myred]
\definebar[myoverstrike:3][myoverstrike][color=mygreen]

Test \myoverstrike{%
    Test \myoverstrike{%
        Test \myoverstrike{Test}
        Test}
    Test}
Test
```

Test ~~Test Test Test Test Test~~ Test

It this the perfect user interface? Probably not, but at least it keeps the implementation quite simple.

The behaviour of the MkIV implementation is roughly the same as in MkII, although now we specify the dimensions and placement in terms of the ratio of the x-height of the current font.

```
Test \overstrike{Test \overstrike{Test \overstrike{Test} Test} Test} Test
\blank
Test \underbar  {Test \underbar  {Test \underbar  {Test} Test} Test} Test
\blank
Test \overbar   {Test \overbar   {Test \overbar   {Test} Test} Test} Test
\blank
Test \underbar  {Test \overbar   {Test \overstrike{Test} Test} Test} Test
\blank
```

Test ~~Test Test Test Test Test~~ Test

Test Test Test Test Test Test Test

Test Test Test Test Test Test Test

Test Test Test ~~Test~~ Test Test Test

As an extra this mechanism can also provide simple backgrounds. The normal background mechanism uses MetaPost and the advantage is that we can use arbitrary shapes but it also carries some limitations. When the development of LuaTeX is a bit further along the road I will add the possibility to use MetaPost shapes in this mechanism.

Before we come to backgrounds, first take a look at these examples:

```
\startbar[underbar]  \input zapf \stopbar \blank
\startbar[underbars] \input zapf \stopbar \blank
```

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.

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.

First notice that it is no problem to span multiple lines and that hyphenation is not influenced at all. Second you can see that continuous rules are also possible. From such a continuous rule to a background is a small step:

```
\definebar
  [backbar]
  [offset=1.5,rulethickness=2.8,color=blue,
   continue=yes,order=background]

\definebar
  [forebar]
  [offset=1.5,rulethickness=2.8,color=blue,
   continue=yes,order=foreground]
```

The following example code looks messy but this has to do with the fact that we want properly spaced sample injection.

```
from here
    \startcolor[white]%
        \startbar[backbar]%
```

```
            \input zapf
            \removeunwantedspaces
        \stopbar
    \stopcolor
\space till here
\blank
from here
    \startbar[forebar]%
        \input zapf
        \removeunwantedspaces
    \stopbar
\space till here
```

from here 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. till here

from here  till here

Watch how we can use the order to hide content. By default rules are drawn on top of the text.

Nice effects can be accomplished with transparencies:

```
\definecolor [tblue] [b=.5,t=.25,a=1]
\setupbars [backbar] [color=tblue]
\setupbars [forebar] [color=tblue]
```

We use as example:

```
from here {\white \backbar{test test}
```

```
    \backbar {nested nested} \backbar{also also}} till here
from here {\white \backbar{test test
    \backbar {nested nested}          also also}} till here
from here {\white \backbar{test test
    \backbar {nested nested}          also also}} till here
```

from here test test nested nested also also till here from here test test nested nested also also till here from here test test nested nested also also till here

The darker nested variant is just the result of two transparent bars on top of each other. We can limit stacking, for instance:

```
\setupbars[backbar][max=1]
\setupbars[forebar][max=1]
```

This gives

from here test test nested nested also also till here from here test test nested nested also also till here from here test test nested nested also also till here

There are currently some limitations that are mostly due to the fact that we use only one attribute for this feature and a change in value triggers another handling. So, we have no real nesting here.

The default commands are defined as follows:

```
\definebar[overstrike]  [method=0,dy= 0.4,offset= 0.5]
\definebar[underbar]    [method=1,dy=-0.4,offset=-0.3]
\definebar[overbar]     [method=1,dy= 0.4,offset= 1.8]

\definebar[overstrikes] [overstrike] [continue=yes]
\definebar[underbars]   [underbar]   [continue=yes]
\definebar[overbars]    [overbar]    [continue=yes]
```

As the implementation is rather non-intrusive you can use bars almost everywhere. You can underbar a whole document but equally well you can stick to fooling around with for instance formulas.

```
\definecolor [tred]   [r=.5,t=.25,a=1]
\definecolor [tgreen] [g=.5,t=.25,a=1]
\definecolor [tblue]  [b=.5,t=.25,a=1]

\definebar [mathred]   [backbar] [color=tred]
```

```
\definebar [mathgreen] [backbar] [color=tgreen]
\definebar [mathblue]  [backbar] [color=tblue]

\startformula
    \mathred{e} = \mathgreen{\white mc} ^ {\mathblue{\white e}}
\stopformula
```

We get:

$$e = mc^e$$

We started this chapter with some words on grouping. In the examples you see no difference between adding bars and for instance applying color. However you need to keep in mind that this is only because behind the screens we keep the current settings along with the attribute. In practice this is only noticeable when you do lots of (local) changes to the settings. Take:

```
{test test test \setupbars[color=red] \underbar{test} test}
```

This results in a local change in settings, which in turn will associate a new attribute to `\underbar`. So, in fact the following underbar becomes a different one than previous underbars. When the page is prepared, the unique attribute value will relate to those settings. Of course there are more mechanisms where such associations take place.

3.4 More to come

Is this all there is? No, as usual the underlying mechanisms can be used for other purposes as well. Take for instance inline notes:

```
According to the wikipedia this is the longest English word:
pneumonoultramicroscopicsilicovolcanoconiosis~\shiftup {other long
words are pseudopseudohypoparathyroidism and
flocci-nauci-nihili-pili-fication}. Of course in languages like Dutch and
German we can make arbitrary long words by pasting words together.
```

This will produce:

According to the wikipedia this is the longest English word: pneumonoultramicroscopicsilicovolcanoconiosis [other long words are pseudopseudohypoparathyroidism and flocci-nauci-nihili-pili-fication]. Of course in languages like Dutch and German we can make arbitrary long words by pasting words together.

I wonder when users really start using such features.

3.5 Summary

Although under the hood the MkIV bar commands are quite different from their MkII counterparts users probably won't notice much difference at first sight. However, the new implementation does not interfere with the par builder and other mechanisms. Plus, it is configurable and it offers more functionality. However, as it is processed rather delayed, side effects might occur that are not foreseen.

So, if you ever notice such unexpected side effects, you know where it might result from: what you asked for is processed much later and by then the circumstances might have changed. If you suspect that it relates to grouping there is a simple remedy: define a new bar command in the document preamble instead of changing properties mid-document. After all, you are supposed to separate rendering and content in the first place.

44 Grouping

4 The font name mess

4.1 Introduction

When TEX came around it shipped with its own fonts. At that moment the TEX font universe was a small and well known territory. The 'only' hassle was that one needed to make sure that the right kind of bitmap was available for the printer.

When other languages than English came into the picture things became more complex as now fonts instances in specific encodings showed up. After a couple of years the by then standardised TEX distributions carried tens of thousands of font files. The reason for this was simple: TEX fonts could only have 256 characters and therefore there were quite some encodings. Also, large CJK fonts could easily have hundreds of metric files per font. Distributions also provide metrics for commercial fonts although I could never use them and as a result have many extra metric files in my personal trees (generated by TEXFONT).[7]

At the input side many problems related to encodings were solved by UNICODE. So, when the more UNICODE aware fonts showed up, it looked like things would become easier. For instance, no longer were choices for encodings needed. Instead one had to choose features and enable languages and scripts and so the problem of the multitude of files was replaced by the necessity to know what some font actually provides. But still, for the average user it can be seen as an improvement.

A rather persistent problem remained, especially for those who want to use different fonts and or need to install fonts on the system that come from elsewhere (either free or commercial): the names used for fonts. You may argue that modern TEX engines and macro packages can make things easier, especially as one can call up fonts by their names instead of their filenames, but actually the problem has worsened. With traditional TEX you definitely get an error when you mistype a filename or call for a font that is not on your system. The more modern TEX's macro packages can provide fallback mechanisms and you can end up with something you didn't ask for.

For years one of the good things of TEX was its stability. If we forget about changes in content, macro packages and/or hyphenation patterns, documents could render more or less the same for years. This is because fonts didn't change. However, now that fonts are more complex, bugs gets fixed and thereby

[7] Distributions like TEXLive have between 50.000 and 100.000 files, but derivatives like the CONTEXT minimals are much smaller.

results can differ. Or, if you use platform fonts, your updated operating system might have new or even different variants. Or, if you access your fonts by fontname, a lookup can resolve differently.

The main reason for this is that fontnames as well as filenames of fonts are highly inconsistent across vendors, within vendors and platforms. As we have to deal with this matter, in MkIV we have several ways to address a font: by filename, by fontname, and by specification. In the next sections I will describe all three.

4.2 Method 1: file

The most robust way to specify what fonts is to be used is the filename. This is done as follows:

```
\definefont[SomeFont][file:lmmono10-regular]
```

A filename lookup is case insensitive and the name you pass is exact. Of course the `file:` prefix (as with any prefix) can be used in font synonyms as well. You may add a suffix, so this is also valid:

```
\definefont[SomeFont][file:lmmono10-regular.otf]
```

By default ConTeXt will first look for an OpenType font so in both cases you will get such a font. But how do you know what the filename is? You can for instance check it out with:

```
mtxrun --script font --list --file --pattern="lm*mono"
```

This reports some information about the file, like the weight, style, width, fontname, filename and optionally the subfont id and a mismatch between the analysed weight and the one mentioned by the font.

```
latinmodernmonolight       light   normal   normal      lmmonolt10regular          lmmonolt10-regular.otf
latinmodernmonoproplight   light   italic   normal      lmmonoproplt10oblique      lmmonoproplt10-oblique.otf
latinmodernmono            normal  normal   normal      lmmono9regular             lmmono9-regular.otf
latinmodernmonoprop        normal  italic   normal      lmmonoprop10oblique        lmmonoprop10-oblique.otf
latinmodernmono            normal  italic   normal      lmmono10italic             lmmono10-italic.otf
latinmodernmono            normal  normal   normal      lmmono8regular             lmmono8-regular.otf
latinmodernmonolightcond   light   italic   condensed   lmmonoltcond10oblique      lmmonoltcond10-oblique.otf
latinmodernmonolight       light   italic   normal      lmmonolt10oblique          lmmonolt10-oblique.otf
latinmodernmonolightcond   light   normal   condensed   lmmonoltcond10regular      lmmonoltcond10-regular.otf
latinmodernmonolight       bold    italic   normal      lmmonolt10boldoblique      lmmonolt10-boldoblique.otf
latinmodernmonocaps        normal  italic   normal      lmmonocaps10oblique        lmmonocaps10-oblique.otf
```

latinmodernmonoproplight	bold	italic	normal	lmmonoproplt10boldoblique	lmmonoproplt10-boldoblique.otf
latinmodernmonolight	bold	normal	normal	lmmonolt10bold	lmmonolt10-bold.otf
latinmodernmonoproplight	bold	normal	normal	lmmonoproplt10bold	lmmonoproplt10-bold.otf
latinmodernmonoslanted	normal	normal	normal	lmmonoslant10regular	lmmonoslant10-regular.otf
latinmodernmono	normal	normal	normal	lmmono12regular	lmmono12-regular.otf
latinmodernmonocaps	normal	normal	normal	lmmonocaps10regular	lmmonocaps10-regular.otf
latinmodernmonoprop	normal	normal	normal	lmmonoprop10regular	lmmonoprop10-regular.otf
latinmodernmono	normal	normal	normal	lmmono10regular	lmmono10-regular.otf
latinmodernmonoproplight	light	normal	normal	lmmonoproplt10regular	lmmonoproplt10-regular.otf

4.3 Method 1: name

Instead of lookup by file, you can also use names. In the font database we store references to the fontname and fullname as well as some composed names from information that comes with the font. This permits rather liberal naming and the main reason is that we can more easily look up fonts. In practice you will use names that are as close to the filename as possible.

```
mtxrun --script font --list --name --pattern="lmmono*regular" --all
```

This gives on my machine:

lmmono10regular	lmmono10regular	lmmono10-regular.otf
lmmono12regular	lmmono12regular	lmmono12-regular.otf
lmmono8regular	lmmono8regular	lmmono8-regular.otf
lmmono9regular	lmmono9regular	lmmono9-regular.otf
lmmonocaps10regular	lmmonocaps10regular	lmmonocaps10-regular.otf
lmmonolt10regular	lmmonolt10regular	lmmonolt10-regular.otf
lmmonoltcond10regular	lmmonoltcond10regular	lmmonoltcond10-regular.otf
lmmonoprop10regular	lmmonoprop10regular	lmmonoprop10-regular.otf
lmmonoproplt10regular	lmmonoproplt10regular	lmmonoproplt10-regular.otf
lmmonoslant10regular	lmmonoslant10regular	lmmonoslant10-regular.otf

It does not show from this list but with name lookups first OpenType fonts are checked and then Type1. In this case there are Type1 variants as well but they are ignored. Fonts are registered under all names that make sense and can be derived from its description. So:

```
mtxrun --script font --list --name --pattern="latinmodern*mono" --all
```

will give:

latinmodernmono	lmmono9regular	lmmono9-regular.otf
latinmodernmonocaps	lmmonocaps10oblique	lmmonocaps10-oblique.otf
latinmodernmonocapsitalic	lmmonocaps10oblique	lmmonocaps10-oblique.otf
latinmodernmonocapsnormal	lmmonocaps10oblique	lmmonocaps10-oblique.otf

latinmodernmonolight	lmmonolt10regular	lmmonolt10-regular.otf
latinmodernmonolightbold	lmmonolt10boldoblique	lmmonolt10-boldoblique.otf
latinmodernmonolightbolditalic	lmmonolt10boldoblique	lmmonolt10-boldoblique.otf
latinmodernmonolightcond	lmmonoltcond10oblique	lmmonoltcond10-oblique.otf
latinmodernmonolightconditalic	lmmonoltcond10oblique	lmmonoltcond10-oblique.otf
latinmodernmonolightcondlight	lmmonoltcond10oblique	lmmonoltcond10-oblique.otf
latinmodernmonolightitalic	lmmonolt10oblique	lmmonolt10-oblique.otf
latinmodernmonolightlight	lmmonolt10regular	lmmonolt10-regular.otf
latinmodernmononormal	lmmono9regular	lmmono9-regular.otf
latinmodernmonoprop	lmmonoprop10oblique	lmmonoprop10-oblique.otf
latinmodernmonopropitalic	lmmonoprop10oblique	lmmonoprop10-oblique.otf
latinmodernmonoproplight	lmmonoproplt10oblique	lmmonoproplt10-oblique.otf
latinmodernmonoproplightbold	lmmonoproplt10boldoblique	lmmonoproplt10-boldoblique.otf
latinmodernmonoproplightbolditalic	lmmonoproplt10boldoblique	lmmonoproplt10-boldoblique.otf
latinmodernmonoproplightitalic	lmmonoproplt10oblique	lmmonoproplt10-oblique.otf
latinmodernmonoproplightlight	lmmonoproplt10oblique	lmmonoproplt10-oblique.otf
latinmodernmonopropnormal	lmmonoprop10oblique	lmmonoprop10-oblique.otf
latinmodernmonoslanted	lmmonoslant10regular	lmmonoslant10-regular.otf
latinmodernmonoslantednormal	lmmonoslant10regular	lmmonoslant10-regular.otf

Watch the 9 point version in this list. It happens that there are 9, 10 and 12 point regular variants but all those extras come in 10 point only. So we get a mix and if you want a specific design size you really have to be more specific. Because one font can be registered with its fontname, fullname etc. it can show up more than once in the list. You get what you ask for.

With this obscurity you might wonder why names make sense as lookups. One advantage is that you can forget about special characters. Also, Latin Modern with its design sizes is probably the worst case. So, although for most fonts a name like the following will work, for Latin Modern it gives one of the design sizes:

```
\definefont[SomeFont][name:latinmodernmonolightbolditalic]
```

But this is quite okay:

```
\definefont[SomeFont][name:lmmonolt10boldoblique]
```

So, in practice this method will work out as well as the file method but you can best check if you get what you want.

4.4 Method 1: spec

We have now arrived at the third method, selecting by means of a specification. This time we take the familyname as starting point (although we have some

fallback mechanisms):

```
\definefont[SomeSerif]           [spec:times]
\definefont[SomeSerifBold]       [spec:times-bold]
\definefont[SomeSerifItalic]     [spec:times-italic]
\definefont[SomeSerifBoldItalic][spec:times-bold-italic]
```

The patterns are of the form:

```
spec:name-weight-style-width
spec:name-weight-style
spec:name-style
```

When only the name is used, it actually boils down to:

```
spec:name-normal-normal-normal
```

So, this is also valid:

```
spec:name-normal-italic-normal
spec:name-normal-normal-condensed
```

Again we can consult the database:

```
mtxrun --script font --list --spec lmmono-normal-italic
```

This prints the following list. The first column is the familyname, the fifth column the fontname:

```
latinmodernmono       normal   italic   normal   lmmono10italic        lmmono10-italic.otf
latinmodernmonoprop   normal   italic   normal   lmmonoprop10oblique   lmmonoprop10-oblique.otf
lmmono10              normal   italic   normal   lmmono10italic        lmtti10.afm
lmmonoprop10          normal   italic   normal   lmmonoprop10oblique   lmvtto10.afm
lmmonocaps10          normal   italic   normal   lmmonocaps10oblique   lmtcso10.afm
latinmodernmonocaps   normal   italic   normal   lmmonocaps10oblique   lmmonocaps10-oblique.otf
```

Watch the OpenType and Type1 mix. As we're just investigating here, the lookup looks at the fontname and not at the familyname. At the TEX end you use the familyname:

```
\definefont[SomeFont][spec:latinmodernmono-normal-italic-normal]
```

So, we have the following ways to access this font:

```
\definefont[SomeFont][file:lmmono10-italic]
\definefont[SomeFont][file:lmmono10-italic.otf]
\definefont[SomeFont][name:lmmono10italic]
\definefont[SomeFont][spec:latinmodernmono-normal-italic-normal]
```

As OpenType fonts are prefered over Type1 there is not much chance of a mixup.

As mentioned in the introduction, qualifications are somewhat inconsistent. Among the weight we find: black, bol, bold, demi, demibold, extrabold, heavy, light, medium, mediumbold, regular, semi, semibold, ultra, ultrabold and ultralight. Styles are: ita, ital, italic, roman, regular, reverseoblique, oblique and slanted. Examples of width are: book, cond, condensed, expanded, normal and thin. Finally we have alternatives which can be anything.

When doing a lookup, some normalizations takes place, with the default always being 'normal'. But still the repertoire is large:

```
helveticaneue medium     normal normal    helveticaneuemedium          HelveticaNeue.ttc index:  0
helveticaneue bold       normal condensed helveticaneuecondensedbold   HelveticaNeue.ttc index:  1
helveticaneue black      normal condensed helveticaneuecondensedblack  HelveticaNeue.ttc index:  2
helveticaneue ultralight italic thin      helveticaneueultralightitalic HelveticaNeue.ttc index:  3
helveticaneue ultralight normal thin      helveticaneueultralight       HelveticaNeue.ttc index:  4
helveticaneue light      italic normal    helveticaneuelightitalic      HelveticaNeue.ttc index:  5
helveticaneue light      normal normal    helveticaneuelight            HelveticaNeue.ttc index:  6
helveticaneue bold       italic normal    helveticaneuebolditalic       HelveticaNeue.ttc index:  7
helveticaneue normal     italic normal    helveticaneueitalic           HelveticaNeue.ttc index:  8
helveticaneue bold       normal normal    helveticaneuebold             HelveticaNeue.ttc index:  9
helveticaneue normal     normal normal    helveticaneue                 HelveticaNeue.ttc index: 10
helveticaneue normal     normal condensed helveticaneuecondensed        hlc_____.afm   conflict: roman
helveticaneue bold       normal condensed helveticaneueboldcond         hlbc____.afm
helveticaneue black      normal normal    helveticaneueblackcond        hlzc____.afm   conflict: normal
helveticaneue black      normal normal    helveticaneueblack            hlbl____.afm   conflict: normal
helveticaneue normal     normal normal    helveticaneueroman            lt_50259.afm   conflict: regular
```

4.5 The font database

In MkIV we use a rather extensive font database which in addition to bare information also contains a couple of hashes. When you use ConTeXt MkIV and install a new font, you have to regenerate the file database. In a next TeX run this will trigger a reload of the font database. Of course you can also force a reload with:

```
mtxrun --script font --reload
```

As a summary we mention a few of the discussed calls of this script:

```
mtxrun --script font --list somename (== --pattern=*somename*)

mtxrun --script font --list --name somename
mtxrun --script font --list --name --pattern=*somename*

mtxrun --script font --list --spec somename
mtxrun --script font --list --spec somename-bold-italic
mtxrun --script font --list --spec --pattern=*somename*
mtxrun --script font --list --spec --filter="fontname=somename"
mtxrun --script font --list --spec --filter="familyname=somename,weight=bold,style=italic,width=condensed"

mtxrun --script font --list --file somename
mtxrun --script font --list --file --pattern=*somename*
```

The lists shown in before depend on what fonts are installed and their version. They might not reflect reality at the time you read this.

4.6 Interfacing

Regular users never deal with the font database directly. However, if you write font loading macros yourself, you can access the database from the TEX end. First we show an example of an entry in the database, in this case TeXGyreTermes Regular.

```
{
    designsize = 100,
    familyname = "texgyretermes",
    filename = "texgyretermes-regular.otf",
    fontname = "texgyretermesregular",
    fontweight = "regular",
    format = "otf",
    fullname = "texgyretermesregular",
    maxsize = 200,
    minsize = 50,
    rawname = "TeXGyreTermes-Regular",
    style = "normal",
    variant = "",
    weight = "normal",
    width = "normal",
}
```

Another example is Helvetica Neue Italic:

```
{
    designsize = 0,
    familyname = "helveticaneue",
    filename = "HelveticaNeue.ttc",
    fontname = "helveticaneueitalic",
    fontweight = "book",
    format = "ttc",
    fullname = "helveticaneueitalic",
    maxsize = 0,
    minsize = 0,
    rawname = "Helvetica Neue Italic",
    style = "italic",
    subfont = 8,
    variant = "",
    weight = "normal",
    width = "normal",
}
```

As you can see, some fields can be meaningless, like the sizes. As using the low level TEX interface assumes some knowledge, we stick here to an example:

```
\def\TestLookup#1%
  {\dolookupfontbyspec{#1}
   pattern: #1, found: \dolookupnoffound
   \blank
   \dorecurse {\dolookupnoffound} {%
     \recurselevel:~\dolookupgetkeyofindex{fontname}{\recurselevel}%
     \quad
   }%
   \blank}

\TestLookup{familyname=helveticaneue}
\TestLookup{familyname=helveticaneue,weight=bold}
\TestLookup{familyname=helveticaneue,weight=bold,style=italic}
```

You can use the following commands:

```
\dolookupfontbyspec    {key=value list}
\dolookupnoffound
\dolookupgetkeyofindex {key}{index}
\dolookupgetkey        {key}
```

First you do a lookup. After that there can be one or more matches and you

can access the fields of each match. What you do with the information is up to yourself.

4.7 A few remarks

The fact that modern TEX engines can access system fonts is promoted as a virtue. The previous sections demonstrated that in practice this does not really free us from a name mess. Of course, when we use a really small TEX tree, and system fonts only, there is not much that can go wrong, but when you have extra fonts installed there can be clashes.

We're better off with filenames than we were in former times when operating systems and media forced distributors to stick to 8 characters in filenames. But that does not guarantee that today's shipments are more consistent. And as there are still some limitations in the length of fontnames, obscure names will be with us for a long time to come.

54 The font name mess

5 The Bidi Dilemma

Here I will introduce a few concepts of bidirectional typesetting. While LuaTEX does a lot automatically, this does not mean that you get a proper bidirectional layout for free. We distinguish a few cases:

- verbatim as used in manuals
- simulating a text editor
- typesetting of text

In addition to this we distinguish two document layouts:

- predominantly left–to–right with some right–to–left snippets
- predominantly right–to–left with some left–to–right snippets

In both cases explicit choices have to be made when defining the layout, programming the style, and coding the content. In this chapter I will stick to verbatim.

In verbatim mode we normally use a monospaced font and no interference with features is to be expected. You get what you've keyed in. Because verbatim is used for illustrative purposes, we need to have predictable output. This is why we have to control the position of the linenumbers as well as the alignment explicitly.

```
\definetyping [XXtyping] [numbering=line]
\definetyping [RLtyping] [align=r2l,numbering=line]
\definetyping [LRtyping] [align=l2r,numbering=line]
```

We use these definitions in the following example:

```
\startLRtyping
At the left!
At the left!
\stopLRtyping

\startRLtyping
At the right!
At the right!
\stopRLtyping

\startalignment[l2r]
\startXXtyping
```

```
At the left!
At the left!
\stopXXtyping
\stopalignment

\startalignment[r2l]
\startXXtyping
At the right!
At the right!
\stopXXtyping
\stopalignment
```

However, we can have a bit more control over the position of the line numbers. As linenumbers are added in a later stage we need to define additional line number classes for this. We show four relevant positions of linenumbers. What setting you use depends on the predominant direction of your document as well as what you want to demonstrate.

```
\definetyping [RLtypingLEFT] [align=r2l,numbering=line]
\definetyping [LRtypingLEFT] [align=l2r,numbering=line]

\setuplinenumbering [RLtypingLEFT] [location=left]
\setuplinenumbering [LRtypingLEFT] [location=left]

\startLRtypingLEFT
At the left!
At the left!
\stopLRtypingLEFT

\startRLtypingLEFT
At the right!
At the right!
\stopRLtypingLEFT
```

When `location` is set to `left`, the line numbers will always be in the left margin, no matter what the text direction is.

```
1  At the left!
2  At the left!

1                                                    !thgir eht tA
2                                                    !thgir eht tA
```

From this it follows that when `location` is set to `right`, the line numbers will always be in the right margin.

```
\definetyping [RLtypingRIGHT] [align=r2l,numbering=line]
\definetyping [LRtypingRIGHT] [align=l2r,numbering=line]

\setuplinenumbering [RLtypingRIGHT] [location=right]
\setuplinenumbering [LRtypingRIGHT] [location=right]

\startLRtypingRIGHT
At the left!
At the left!
\stopLRtypingRIGHT

\startRLtypingRIGHT
At the right!
At the right!
\stopRLtypingRIGHT
```

Again, the text direction is not influencing the placement.

```
At the left!                                                    1
At the left!                                                    2

                                                !thgir eht tA   1
                                                !thgir eht tA   2
```

The next two cases *do* obey to the text direction. When set to `begin`, the location will be at the beginning of the line.

```
\definetyping [RLtypingBEGIN] [align=r2l,numbering=line]
\definetyping [LRtypingBEGIN] [align=l2r,numbering=line]

\setuplinenumbering [RLtypingBEGIN] [location=begin]
\setuplinenumbering [LRtypingBEGIN] [location=begin]

\startLRtypingBEGIN
At the left!
At the left!
\stopLRtypingBEGIN

\startRLtypingBEGIN
At the right!
At the right!
```

```
\stopRLtypingBEGIN
```

When typesetting a paragraph from right to left, the beginning of the line is at the right margin.

```
1   At the left!
2   At the left!
```

```
                                                    !thgir eht tA    1
                                                    !thgir eht tA    2
```

Consequently we get the opposite result when we set `location` to `end`.

```
\definetyping [RLtypingEND] [align=r2l,numbering=line]
\definetyping [LRtypingEND] [align=l2r,numbering=line]

\setuplinenumbering [RLtypingEND] [location=end]
\setuplinenumbering [LRtypingEND] [location=end]

\startLRtypingEND
At the left!
At the left!
\stopLRtypingEND

\startRLtypingEND
At the right!
At the right!
\stopRLtypingEND
```

This time we get complementary results:

```
At the left!                                                    1
At the left!                                                    2
```

```
                                                    !thgir eht tA
                                                    !thgir eht tA
```

It will be clear that when we are writing a manual where we mix example code with real right to left text some care goes into setting up the verbatim environments. And this is just one of the aspects you have to deal with in a bidirectional document layout.

6 Deeply nested notes

6.1 Introduction

One of the mechanisms that is not on a users retina when he or she starts using TEX is 'inserts'. An insert is material that is entered at one point but will appear somewhere else in the output. Footnotes for instance can be implemented using inserts. You create a reference symbol in the running text and put note text at the bottom of the page or at the end of a chapter or document. But as you don't want to do that moving around of notes yourself TEX provides macro writers with the inserts mechanism that will do some of the housekeeping. Inserts are quite clever in the sense that they are taken into account when TEX splits off a page. A single insert can even be split over two or more pages.

Other examples of inserts are floats that move to the top or bottom of the page depending on requirements and/or available space. Of course the macro package is responsible for packaging such a float (for instance an image) but by finally putting it in an insert TEX itself will attempt to deal with accumulated floats and help you move kept over floats to following pages. When the page is finally assembled (in the output routine) the inserts for that page become available and can be put at the spot where they belong. In the process TEX has made sure that we have the right amount of space available.

However, let's get back to notes. In ConTEXt we can have many variants of them, each taken care of by its own class of inserts. This works quite well, as long as a note is visible for TEX which means as much as: ends up in the main page flow. Consider the following situation:

```
before \footnote{the note} after
```

When the text is typeset, a symbol is placed directly after the word 'before' and the note itself ends up at the bottom of the page. It also works when we wrap the text in an horizontal box:

```
\hbox{before \footnote{the note} after}
```

But it fails as soon as we go further:

```
\hbox{\hbox{before \footnote{the note} after}}
```

Here we get the reference but no note. This also fails:

```
\vbox{before \footnote{the note} after}
```

Can you imagine what happens if we do the following?

```
\starttabulate
\NC knuth \NC test \footnote{knuth} \input knuth \NC \NR
\NC tufte \NC test \footnote{tufte} \input tufte \NC \NR
\NC ward  \NC test \footnote{ward}  \input ward  \NC \NR
\stoptabulate
```

This mechanism uses alignments as well as quite some boxes. The paragraphs are nicely split over pages but still appear as boxes to TEX which make inserts invisible. Only the three symbols would remain visible. But because in CONTEXT we know when notes tend to disappear, we take some provisions, and contrary to what you might expect the notes actually do show up. However, they are flushed in such a way that they end up on the page where the table ends. Normally this is no big deal as we will often use local notes that end up at the end of the table instead of the bottom of the page, but still.

The mechanism to deal with notes in CONTEXT is somewhat complex at the source code level. To mention a few properties we have to deal with:

- Notes are collected and can be accessed any time.
- Notes are flushed either directly or delayed.
- Notes can be placed anywhere, any time, perhaps in subsets.
- Notes can be associated to lines in paragraphs.
- Notes can be placed several times with different layouts.

So, we have some control over flushing and placement, but real synchronization between for instance table entries having notes and the note content ending up on the same page is impossible.

In the LUATEX team we have been discussing more control over inserts and we will definitely deal with that in upcoming releases as more control is needed for complex multi-column document layouts. But as we have some other priorities these extensions have to wait.

As a prelude to them I experimented a bit with making these deeply buried inserts visible. Of course I use LUA for this as TEX itself does not provide the kind of access we need for this kind of of manipulations.

6.2 Deep down inside

Say that we have the following boxed footnote. How does that end up in LUATEX?

```
\vbox{a\footnote{b}c}
```

Actually it depends on the macro package but the principles remain the same. In LuaTeX 0.50 and the ConTeXt version used at the time of this writing we get (nested) linked list that prints as follows:

```
<node   26 <  862 >  nil : vlist 0>
  <node  401 <  838 >  507 : hlist 1>
    <node   30 <  611 >  580 : whatsit 6>
    <node  611 <  580 >  493 : hlist 0>
    <node  580 <  493 >  653 : glyph 256>
    <node  493 <  653 >  797 : penalty 0>
    <node  653 <  797 >  424 : kern 1>
    <node  797 <  424 >  826 : hlist 2>
      <node  445 <  563 >  nil : hlist 2>
        <node  420 <  817 >  821 : whatsit 35>
        <node  817 <  821 >  nil : glyph 256>
    <node  507 <  826 > 1272 : kern 1>
    <node  826 < 1272 > 1333 : glyph 256>
    <node 1272 < 1333 >  830 : penalty 0>
    <node 1333 <  830 >  888 : glue 15>
    <node  830 <  888 >  nil : glue 9>
  <node  838 <  507 >  nil : ins 131>
```

The numbers are internal references to the node memory pool. Each line represents a node:

```
<node prev_index < index > next_index : type subtype>
```

The whatsits carry directional information and the deeply nested hlist is the note symbol. If we forget about whatsits, kerns and penalties, we can simplify this listing to:

```
<node   26 <  862 >  nil : vlist 0>
  <node  401 <  838 >  507 : hlist 1>
    <node  580 <  493 >  653 : glyph 256>
    <node  797 <  424 >  826 : hlist 2>
      <node  445 <  563 >  nil : hlist 2>
        <node  817 <  821 >  nil : glyph 256>
    <node  826 < 1272 > 1333 : glyph 256>
  <node  838 <  507 >  nil : ins 131>
```

So, we have a vlist (the `\vbox`), which has one line being a hlist. Inside we have a glyph (the 'a') followed by the raised symbol (the '[1]') and next comes the

second glyph (the 'b'). But watch how the insert ends up at the end of the line. Although the insert will not show up in the document, it sits there waiting to be used. So we have:

```
<node   26 <  862 >  nil : vlist 0>
  <node  401 <  838 >  507 : hlist 1>
  <node  838 <  507 >  nil : ins 131>
```

but we need:

```
<node   26 <  862 >  nil : vlist 0>
  <node  401 <  838 >  507 : hlist 1>
<node  838 <  507 >  nil : ins 131>
```

Now, we could use the fact that inserts end up at the end of the line, but as we need to recursively identify them anyway, we cannot actually use this fact to optimize the code.

In case you wonder how multiple inserts look like, here is an example:

```
\vbox{a\footnote{b}\footnote{c}d}
```

This boils down to:

```
<node   26 < 1324 >  nil : vlist 0>
  <node  401 < 1348 >  507 : hlist 1>
  <node 1348 <  507 >  457 : ins 131>
  <node  507 <  457 >  nil : ins 131>
```

In case you wonder what more can end up at the end, vertically adjusted material (`\vadjust`) as well as marks (`\mark`) also get that treatment.

```
\vbox{a\footnote{b}\vadjust{c}\footnote{d}e\mark{f}}
```

As you see, we start with the line itself, followed by a mixture of inserts and vertically adjusted content (that will be placed before that line). This trace also shows the list 2 levels deep.

```
<node   26 < 1324 >  nil : vlist 0>
  <node  401 < 1348 >  507 : hlist 1>
  <node 1348 <  507 >  862 : ins 131>
  <node  507 <  862 >  240 : hlist 1>
  <node  862 <  240 > 2288 : ins 131>
  <node  240 < 2288 >  nil : mark 0>
```

Currently vadjust nodes have the same subtype as an ordinary hlist but in LuaTeX versions beyond 0.50 they will have a dedicated subtype.

We can summarize the pattern of one 'line' in a vertical list as:

```
[hlist][insert|mark|vadjust]*[penalty|glue]+
```

In case you wonder what happens with for instance specials, literals (and other whatits): these end up in the hlist that holds the line. Only inserts, marks and vadjusts migrate to the outer level, but as they stay inside the vlist, they are not visible to the page builder unless we're dealing with the main vertical list. Compare:

```
this is a regular paragraph possibly with inserts and they
will be visible as the lines are appended to the main
vertical list \par
```

with:

```
but \vbox {this is a nested paragraph where inserts will
stay with the box} and not migrate here \par
```

So much for the details; let's move on the how we can get around this phenomenon.

6.3 Some LUATEX magic

The following code is just the first variant I made and ConTeXt ships with a more extensive variant. Also, in ConTeXt this is part of a larger suite of manipulative actions but it does not make much sense (at least not now) to discuss this framework here.

We start with defining a couple of convenient shortcuts.

```
local hlist = node.id('hlist')
local vlist = node.id('vlist')
local ins   = node.id('ins')
```

We can write a more compact solution but splitting up the functionality better shows what we're doing. The main migration function hooks into the callback `build_page`. Contrary to other callbacks that do phases in building lists and pages this callback does not expect the head of a list as argument. Instead, we operate directly on the additions to the main vertical list which is accessible as

```
tex.lists.contrib_head.

local deal_with_inserts -- forward reference

local function migrate_inserts(where)
    local current = tex.lists.contrib_head
    while current do
        local id = current.id
        if id == vlist or id == hlist then
            current = deal_with_inserts(current)
        end
        current = current.next
    end
end

callback.register('buildpage_filter',migrate_inserts)
```

So, effectively we scan for vertical and horizontal lists and deal with embedded inserts when we find them. In ConTeXt the migratory function is just one of the functions that is applied to this filter.

We locate inserts and collect them in a list with `first` and `last` as head and tail and do so recursively. When we have run into inserts we insert them after the horizontal or vertical list that had embedded them.

```
local locate -- forward reference

deal_with_inserts = function(head)
    local h, first, last = head.list, nil, nil
    while h do
        local id = h.id
        if id == vlist or id == hlist then
            h, first, last = locate(h,first,last)
        end
        h = h.next
    end
    if first then
        local n = head.next
        head.next = first
        first.prev = head
        if n then
            last.next = n
            n.prev = last
```

```
        end
        return last
    else
        return head
    end
end
```

The `locate` function removes inserts and adds them to a new list, that is passed on down in recursive calls and eventually is returned back to the caller.

```
locate = function(head,first,last)
    local current = head
    while current do
        local id = current.id
        if id == vlist or id == hlist then
            current.list, first, last = locate(current.list,first,last)
            current = current.next
        elseif id == ins then
            local insert = current
            head, current = node.remove(head,current)
            insert.next = nil
            if first then
                insert.prev = last
                last.next = insert
            else
                insert.prev = nil
                first = insert
            end
            last = insert
        else
            current = current.next
        end
    end
    return head, first, last
end
```

As we can encounter the content several times in a row, it makes sense to mark already processed inserts. This can for instance be done by setting an attribute. Of course one has to make sure that this attribute is not used elsewhere.

```
if not node.has_attribute(current,8061) then
    node.set_attribute(current,8061,1)
    current = deal_with_inserts(current)
```

```
end
```

or integrated:

```
local has_attribute = node.has_attribute
local set_attribute = node.set_attribute

local function migrate_inserts(where)
    local current = tex.lists.contrib_head
    while current do
        local id = current.id
        if id == vlist or id == hlist then
            if has_attribute(current,8061) then
                -- maybe some tracing message
            else
                set_attribute(current,8061,1)
                current = deal_with_inserts(current)
            end
        end
        current = current.next
    end
end

callback.register('buildpage_filter',migrate_inserts)
```

6.4 A few remarks

Surprisingly, the amount of code needed for insert migration is not that large. This makes one wonder why TEX does not provide this feature itself as it could have saved macro writers quite some time and headaches. Performance can be a reason, unpredictable usage and side effects might be another. Only one person knows the answer.

In ConTEXt this mechanism is built in and it can be enabled by saying:

```
\automigrateinserts
\automigratemarks
```

As you can see here, we can also migrate marks. Future versions of ConTEXt will do this automatically and also provide some control over what classes of inserts are moved around. We will probably overhaul the note handling mechanism a few more times anyway as LuaTEX evolves and the demands from critical editions that use many kind of notes raise.

6.5 Summary of code

The following code should work in plain TEX:

```
\directlua 0 {
local hlist         = node.id('hlist')
local vlist         = node.id('vlist')
local ins           = node.id('ins')
local has_attribute = node.has_attribute
local set_attribute = node.set_attribute

local status = 8061

local function locate(head,first,last)
    local current = head
    while current do
        local id = current.id
        if id == vlist or id == hlist then
            current.list, first, last = locate(current.list,first,last)
            current = current.next
        elseif id == ins then
            local insert = current
            head, current = node.remove(head,current)
            insert.next = nil
            if first then
                insert.prev, last.next = last, insert
            else
                insert.prev, first = nil, insert
            end
            last = insert
        else
            current = current.next
        end
    end
    return head, first, last
end

local function migrate_inserts(where)
    local current = tex.lists.contrib_head
    while current do
        local id = current.id
        if id == vlist or id == hlist and
                not has_attribute(current,status) then
```

```
            set_attribute(current,status,1)
            local h, first, last = current.list, nil, nil
            while h do
                local id = h.id
                if id == vlist or id == hlist then
                    h, first, last = locate(h,first,last)
                end
                h = h.next
            end
            if first then
                local n = current.next
                if n then
                    last.next, n.prev = n, last
                end
                current.next, first.prev = first, current
                current = last
            end
        end
        current = current.next
    end
end

callback.register('buildpage_filter', migrate_inserts)
}
```

Alternatively you can put the code in a file and load that with:

```
\directlua {require "luatex-inserts.lua"}
```

A simple plain test is:

```
\vbox{a\footnote{1}{1}b}
\hbox{a\footnote{2}{2}b}
```

The first footnote only shows up when we have hooked our migrator into the callback. A not that bad result for 60 lines of LUA code.

7 Upto ConTEXt MkVI

7.1 Introduction

No, this is not a typo: MkVI is the name of upcoming functionality but with an experimental character. It is also a playground. Therefore this is not the final story.

7.2 Defining macros

When you define macros in TEX, you use the # to indicate variables. So, you code can end up with the following:

```
\def\MyTest#1#2#3#4%
  {\dontleavehmode
   \dostepwiserecurse{#1}{#2}{#3}
     {\ifnum\recurselevel>#1 \space,\fi
      \recurselevel: #4\space}%
   .\par}
```

This macro is called with 4 arguments:

```
\MyTest{3}{8}{1}{Hi}
```

However, using numbers as variable identifiers might not have your preference. It makes perfect sense if you keep in mind that TEX supports delimited arguments using arbitrary characters. But in practice, and especially in ConTEXt we use only a few well defined variants. This is why you can also imagine:

```
\def\MyTest#first#last#step#text%
  {\dontleavehmode
   \dostepwiserecurse{#first}{#last}{#step}
     {\ifnum\recurselevel>#first \space,\fi
      \recurselevel: #text}%
   .\par}
```

In order for this to work, you need to give your file the suffix `mkvi` or you need to put a directive on the first line:

```
% macros=mkvi
```

You can of course use delimited arguments as well, given that the delimiters

are not letters.

```
\def\TestOne[#1]%
  {this is: #1}

\def\TestTwo#some%
  {this is: #some}

\def\TestThree[#whatever][#more]%
  {this is: #more and #whatever}

\def\TestFour[#one]#two%
  {\def\TestFive[#alpha][#one]%
     {#one, #two, #alpha}}
```

You can also use the following variant which is already present for a while but not that much advertised. This method ignores all spaces in definitions so if you need one, you have to use `\space`.

```
\starttexdefinition TestSix #oeps

  here: #oeps

\stoptexdefinition
```

These commands work as expected:

```
\startlines
  \TestOne  [one]
  \TestTwo  {one}
  \TestThree[one][two]
  \TestFour [one]{two}
  \TestFive [one][two]
  \TestSix  {one}
\stoplines
```

this is: one
this is: one
this is: two and one
two, two, one
here: one

You can use buffers to collect definitions. In that case you can force prepro-

cessing of the buffer with `\mkvibuffer[name]`.

7.3 Implementation

This functionality is not hard codes in the LuaTEX engine as this is not needed at all. We just preprocess the file before it gets loaded and this is something that is relatively easy to implement. Already early in the development of LuaTEX we have decided that instead of hard coding solutions, opening up makes more sense.

One of the first mechanisms that were opened up was file IO. This means that when a file is opened, you can decide to intercept lines and process them before passing them to the traditional built in input parser. The user can be completely unaware of this. In fact, as LuaTEX only accepts UTF-8 preprocessing will likely happen already when other input encodings are used.

The following helper functions are available:

```
local result = resolvers.macros.preprocessed(str)
```

This function returns a string with all named parameters replaced.

```
resolvers.macros.convertfile(oldname,newname)
```

This function converts a file into a new one.

```
local result = resolvers.macros.processmkvi(str,filename)
```

This function converts the string but only if the suffix of the filename is `mkvi` or when the first line of the string is a comment line containing `macros=mkvi`. Otherwise the original string is returned. The filename is optional.

7.4 A few details

Imagine that you want to do this:

```
\def\test#1{before#1after}
```

When we use names this could look like:

```
\def\test#inbetween{before#inbetweenafter}
```

and that is not going to work out well. We could be more liberal with spaces, like

```
\def\test #inbetween {before #inbetween after}
```

but then getting spaces in the output before or after variables would get more complex. However, there is a way out:

```
\def\test#inbetween{before#{inbetween}after}
```

As the sequence `#{` has a rather low probablility of showing up in a TEX source file, this kind of escaping is part of the game. So, all the following cases are valid:

```
\def\test#oeps{... #oeps ...}
\def\test#oeps{... #{oeps} ...}
\def\test#{main:oeps}{... #{main:oeps} ...}
\def\test#{oeps:1}{... #{oeps:1} ...}
\def\test#{oeps}{... #oeps ...}
```

When you use the braced variant, all characters except braces are acceptable as name, optherwise only lowercase and uppercase characters are permitted.

Normally TEX uses a couple of special tokens like `^` and `_`. In a macro definition file you can avoid these by using primitives:

```
&   \aligntab
#   \alignmark
^   \Usuperscript
_   \Usubscript
$   \Ustartmath
$   \Ustopmath
$$  \Ustartdisplaymath
$$  \Ustopdisplaymath
```

Especially the `aligntab` is worth noticing: using that one directly in a macro definition can result in unwanted replacements, depending whether a match can be found. In practice the following works out well

```
\def\test#oeps{test:#oeps \halign{##\cr #oeps\cr}}
```

You can use UTF-8 characters as well. For practical reasons this is only possible with the braced variant.

```
\def\blä#{blá}{blà:#{blá}}
```

There will probably be more features in future versions but each of them needs careful consideration in order to prevent interferences.

7.5 Utilities

There is currently one utility (or in fact an option to an existing utility):

```
mtxrun --script interface --preprocess whatever.mkvi
```

This will convert the given file(s) to new ones, with the default suffix `tex`. Existing files will not be overwritten unless `---force` is given. You can also force another suffix:

```
mtxrun --script interface --preprocess whatever.mkvi --suffix=mkiv
```

A rather plain module `luatex-preprocessor.lua` is provided for other usage. That variant provides a somewhat simplified version.

Given that you have a `luatex-plain` format you can run:

```
luatex --fmt=luatex-plain luatex-preprocessor-test.tex
```

Such a plain format can be made with:

```
luatex --ini luatex-plain
```

You probably need to move the format to a proper location in your TEX tree.

8 Backend code

8.1 Introduction

In ConTeXt we've always separated the backend code in so called driver files. This means that in the code related to typesetting only calls to the API take place, and no backend specific code is to be used. That way we can support backend like dvipsone (and dviwindo), dvips, acrobat, pdftex and dvipdfmx with one interface. A simular model is used in MkIV although at the moment we only have one backend: PDF.[8]

Some ConTeXt users like to add their own PDF specific code to their styles or modules. However, such extensions can interfere with existing code, especially when resources are involved. This has to be done via the official helper macros.

In the next sections an overview will be given of the current approach. There are still quite some rough edges but these will be polished as soon as the backend code is more isolated in LuaTeX itself.

8.2 Structure

A PDF file is a tree of indirect objects. Each object has a number and the file contains a table (or multiple tables) that relates these numbers to positions in a file (or position in a compressed object stream). That way a file can be viewed without reading all data: a viewer only loads what is needed.

```
1 0 obj <<
    /Name (test) /Address 2 0 R
>>
2 0 obj [
   (Main Street) (24) (postal code) (MyPlace)
]
```

For the sake of the discussion we consider strings like `(test)` also to be objects. In the next table we list what we can encounter in a PDF file. There can be indirect objects in which case a reference is used (`2 0 R`) and direct ones.

type	form	meaning
constant	`/...`	A symbol (prescribed string).

[8] At this moment we only support the native PDF backend but future versions might support XML (HTML) output as well.

string	`(...)`	A sequence of characters in pdfdoc encoding
unicode	`<...>`	A sequence of characters in utf16 encoding
number	`3.1415`	A number constant.
boolean	`true/false`	A boolean constant.
reference	`N 0 R`	A reference to an object
dictionary	`<< ... >>`	A collection of key value pairs where the value itself is an (indirect) object.
array	`[ ... ]`	A list of objects or references to objects.
stream		A sequence of bytes either or not packaged with a dictionary that contains descriptive data.
xform		A special kind of object containing an reusable blob of data, for example an image.

While writing additional backend code, we mostly create dictionaries.

```
<< /Name (test) /Address 2 0 R >>
```

In this case the indirect object can look like:

```
[ (Main Street) (24) (postal code) (MyPlace) ]
```

It all starts in the document's root object. From there we access the page tree and resources. Each page carries its own resource information which makes random access easier. A page has a page stream and there we find the to be rendered content as a mixture of (UNICODE) strings and special drawing and rendering operators. Here we will not discuss them as they are mostly generated by the engine itself or dedicated subsystems like the MetaPost converter. There we use literal or `\latelua` whatsits to inject code into the current stream.

In the CONTEXT MKII backend drivers code you will see objects in their verbose form. The content is passed on using special primitives, like `\pdfobj`, `\pdfannot`, `\pdfcatalog`, etc. In MKIV no such primitives are used. In fact, some of them are overloaded to do nothing at all. In the LUA backend code you will find function calls like:

```
local d = lpdf.dictionary {
    Name    = lpdf.string("test"),
    Address = lpdf.array {
        "Main Street", "24", "postal code", "MyPlace",
    }
}
```

Equaly valid is:

```
local d = lpdf.dictionary()
d.Name = "test"
```

Eventually the object will end up in the file using calls like:

```
local r = pdf.immediateobj(tostring(d))
```

or using the wrapper (which permits tracing):

```
local r = lpdf.flushobject(d)
```

The object content will be serialized according to the formal specification so the proper `<< >>` etc. are added. If you want the content instead you can use a function call:

```
local dict = d()
```

An example of using references is:

```
local a = lpdf.array {
    "Main Street", "24", "postal code", "MyPlace",
}
local d = lpdf.dictionary {
    Name    = lpdf.string("test"),
    Address = lpdf.reference(a),
}
local r = lpdf.flushobject(d)
```

We have the following creators. Their arguments are optional.

function	**optional parameter**
`lpdf.dictionary`	hash with key/values
`lpdf.array`	indexed table of objects
`lpdf.unicode`	string
`lpdf.string`	string
`lpdf.number`	number
`lpdf.constant`	string
`lpdf.null`	
`lpdf.boolean`	boolean
`lpdf.reference`	string
`lpdf.verbose`	indexed table of strings

Flushing objects is done with:

```
lpdf.flushobject(obj)
```

Reserving object is or course possible and done with:

```
local r = lpdf.reserveobject()
```

Such an object is flushed with:

```
lpdf.flushobject(r,obj)
```

We also support named objects:

```
lpdf.reserveobject("myobject")

lpdf.flushobject("myobject",obj)
```

8.3 Resources

While LuaTEX itself will embed all resources related to regular typesetting, MkIV has to take care of embedding those related to special tricks, like annotations, spot colors, layers, shades, transparencies, metadata, etc. If you ever took a look in the MkII `spec-*` files you might have gotten the impression that it quickly becomes messy. The code there is actually rather old and evolved in sync with the PDF format as well as pdfTEX and dvipdfmx maturing to their current state. As a result we have a dedicated object referencing model that sometimes results in multiple passes due to forward references. We could have gotten away from that with the latest versions of pdfTEX as it provides means to reserve object numbers but it makes not much sense to do that now that MkII is frozen.

Because third party modules (like tikz) also can add resources like in MkII using an API that makes sure that no interference takes place. Think of macros like:

```
\pdfbackendsetcatalog       {key}{string}
\pdfbackendsetinfo          {key}{string}
\pdfbackendsetname          {key}{string}

\pdfbackendsetpageattribute {key}{string}
\pdfbackendsetpagesattribute{key}{string}
\pdfbackendsetpageresource  {key}{string}

\pdfbackendsetextgstate     {key}{pdfdata}
\pdfbackendsetcolorspace    {key}{pdfdata}
```

```
\pdfbackendsetpattern       {key}{pdfdata}
\pdfbackendsetshade         {key}{pdfdata}
```

One is free to use the LUA interface instead, as there one has more possibilities. The names are similar, like:

```
lpdf.addtoinfo(key,anything_valid_pdf)
```

At the time of this writing (LUATEX .50) there are still places where TEX and LUA code is interwoven in a non optimal way, but that will change in the future as the backend is completely separated and we can do more TEX trickery at the LUA end.

Also, currently we expose more of the backend code than we like and future versions will have a more restricted access. The following function will stay public:

```
lpdf.addtopageresources  (key,value)
lpdf.addtopageattributes (key,value)
lpdf.addtopagesattributes(key,value)

lpdf.adddocumentextgstate(key,value)
lpdf.adddocumentcolorspac(key,value)
lpdf.adddocumentpattern  (key,value)
lpdf.adddocumentshade    (key,value)

lpdf.addtocatalog        (key,value)
lpdf.addtoinfo           (key,value)
lpdf.addtonames          (key,value)
```

There are several tracing options built in and some more will be added in due time:

```
\enabletrackers
  [backend.finalizers,
   backend.resources,
   backend.objects,
   backend.detail]
```

As with all trackers you can also pass them on the command line, for example:

```
context --trackers=backend.* yourfile
```

The reference related backend mechanisms have their own trackers.

8.4 Transformations

There is at the time of this writing still some backend related code at the TEX end that needs a cleanup. Most noticeable is the code that deals with transformations (like scaling). At some moment in pdfTEX a primitive was introduced but it was not completely covering the transform matrix so we never used it. In LuaTEX we will come up with a better mechanism. Till that moment we stick to the MkII method.

8.5 Annotations

The Lua based backend of MkIV is not so much less code, but definitely cleaner. The reason why there is quite some code is because in ConTEXt we also handle annotations and destinations in Lua. In other words: TEX is not bothered by the backend any more. We could make that split without too much impact as we never depended on pdfTEX hyperlink related features and used generic annotations instead. It's for that reason that ConTEXt has always been able to nest hyperlinks and have annotations with a chain of actions.

Another reason for doing it all at the Lua end is that as in MkII we have to deal with the rather hybrid cross reference mechanisms which uses a sort of language and parsing this is also easier at the Lua end. Think of:

```
\definereference[somesound][StartSound(attention)]

\at {just some page} [someplace,somesound,StartMovie(somemovie)]
```

We parse the specification expanding shortcuts when needed, create an action chain, make sure that the movie related resources are taken care of (normally the movie itself will be a figure), and turn the three words into hyperlinks. As this all happens in Lua we have less TEX code. Contrary to what you might expect, the Lua code is not that much faster as the MkII TEX code is rather optimized.

Special features like JavaScript as well as widgets (and forms) are also reimplemented. Support for JavaScript is not that complex at all, but as in ConTEXt we can organize scripts in collections and have automatic inclusion of used functions, still some code is needed. As we now do this in Lua we use less TEX memory. Reimplementing widgets took a bit more work as I used the opportunity to remove hacks for older viewers. As support for widgets is somewhat

instable in viewers quite some testing was needed, especially because we keep supporting cloned and copied fields (resulting in widget trees).

An interesting complication with widgets is that each instance can have a lot of properties and as we want to be able to use thousands of them in one document, each with different properties, we have efficient storage in MkII and want to do the same in Lua. Most code at the TEX end is related to passing all those options.

You could use the Lua functions that relate to annotations etc. but normally you will use the regular ConTEXt user interface. For practical reasons, the backend code is grouped in several tables:

The `backends` table has subtables for each backend and currently there is only one: `pdf`. Each backend provides tables itself. In the `codeinjections` namespace we collect functions that don't interfere with the typesetting or typeset result, like inserting all kind of resources (movies, attachment, etc.), widget related functionality, and in fact everything that does not fit into the other categories. In `nodeinjections` we organize functions that inject literal pdf code in the nodelist which then ends up in the pdf stream: color, layers, etc. The `registrations` table is reserved for functions related to resources that result from node injections: spot colors, transparencies, etc. Once the backend code is finished we might come up with another organization. No matter what we end up with, the way the `backends` table is supposed to be organized determines the api and those who have seen the MkII backend code will recognize some of it.

8.6 Metadata

We always had the opportunity to set the information fields in a pdf but standardization forces us to add these large verbose metadata blobs. As this blob is coded in xml we use the built in xml parser to fill a template. Thanks to extensive testing and research by Peter Rolf we now have a rather complete support for pdf/x related demands. This will definitely evolve with the advance of the pdf specification. You can replace the information with your own but we suggest that you stay away from this metadata mess as far as possible.

8.7 Helpers

If you look into the `lpdf-*.lua` files you will find more functions. Some are public helpers, like:

`lpdf.toeight(str)` returns `(string)`
`lpdf.tosixteen(str)` returns `<utf16 sequence>`

An example of another public function is:

```
lpdf.sharedobj(content)
```

This one flushes the object and returns the object number. Already defined objects are reused. In addition to this code driven optimization, some other optimization and reuse takes place but all that happens without user intervention.

9 Callbacks

9.1 Introduction

Callbacks are the means to extend the basic TEX engine's functionality in LuaTEX and ConTEXt MkIV uses them extensively. Although the interface is still in development we see users popping in their own functionality and although there is nothing wrong with that, it can open a can of worms.

It is for this reason that from now on we protect the MkIV callbacks from being overloaded. For those who still want to add their own code some hooks are provided. Here we will address some of these issues.

9.2 Actions

There are already quite some callbacks and we use most of them. In the following list the callbacks tagged with `enabled` are used and frozen, the ones tagged `disabled` are blocked and never used, while the ones tagged `undefined` are yet unused.

`append_to_vlist_filter`	`undefined`	
`buildpage_filter`	`enabled`	vertical spacing etc (mvl)
`char_exists`	`undefined`	
`contribute_filter`	`undefined`	
`define_font`	`enabled`	definition of fonts (tfmdata preparation)
`find_cidmap_file`	`enabled`	find file using resolver
`find_data_file`	`enabled`	find file using resolver
`find_enc_file`	`enabled`	find file using resolver
`find_font_file`	`enabled`	find file using resolver
`find_format_file`	`enabled`	find file using resolver
`find_image_file`	`enabled`	find file using resolver
`find_map_file`	`enabled`	find file using resolver
`find_opentype_file`	`enabled`	find file using resolver
`find_output_file`	`enabled`	find file using resolver
`find_pk_file`	`enabled`	find file using resolver
`find_read_file`	`enabled`	find file using resolver
`find_sfd_file`	`enabled`	find file using resolver
`find_truetype_file`	`enabled`	find file using resolver
`find_type1_file`	`enabled`	find file using resolver
`find_vf_file`	`enabled`	find file using resolver
`find_write_file`	`enabled`	find file using resolver
`finish_pdffile`	`enabled`	

finish_pdfpage	enabled	
hpack_filter	enabled	all kind of horizontal manipulations (before hbox creation)
hpack_quality	undefined	
hyphenate	disabled	normal hyphenation routine, called elsewhere
insert_local_par	undefined	
kerning	disabled	normal kerning routine, called elsewhere
ligaturing	disabled	normal ligaturing routine, called elsewhere
linebreak_filter	enabled	breaking paragraps into lines
mlist_to_hlist	enabled	preprocessing math list
open_read_file	enabled	open file for reading
post_linebreak_filter	enabled	all kind of horizontal manipulations (after par break)
pre_dump	enabled	lua related finalizers called before we dump the format
pre_linebreak_filter	enabled	all kind of horizontal manipulations (before par break)
pre_output_filter	undefined	
process_input_buffer	disabled	actions performed when reading data
process_jobname	undefined	
process_output_buffer	disabled	actions performed when writing data
process_rule	enabled	
read_cidmap_file	undefined	read file at once
read_data_file	enabled	read file at once
read_enc_file	enabled	read file at once
read_font_file	enabled	read file at once
read_map_file	enabled	read file at once
read_opentype_file	undefined	read file at once
read_pk_file	enabled	read file at once
read_sfd_file	enabled	read file at once
read_truetype_file	undefined	read file at once
read_type1_file	undefined	read file at once
read_vf_file	enabled	read file at once
show_error_hook	enabled	
show_error_message	enabled	
show_lua_error_hook	enabled	
show_warning_message	enabled	
start_file	enabled	
start_page_number	enabled	actions performed at the beginning of a shipout
start_run	enabled	actions performed at the beginning of a run

stop_file	enabled	
stop_page_number	enabled	actions performed at the end of a shipout
stop_run	enabled	actions performed at the end of a run
vpack_filter	enabled	vertical spacing etc
vpack_quality	undefined	

You can be rather sure that we will eventually use all callbacks one way or the other. Also, some callbacks are only set when certain functionality is enabled.

It may sound somewhat harsh but if users kick in their own code, we cannot guarantee ConTeXt's behaviour any more and support becomes a pain. If you really need to use a callback yourself, you should use one of the hooks and make sure that you return the right values.

The exact working of the callback handler is not something we need to bother users with so we stick to a simple description. The next list is not definitive and evolves. For instance we might at some point decide to add more granularity.

We only open up some of the node list related callbacks. All callbacks related to file handling, font definition and housekeeping are frozen. Most if the mechanisms that use these callbacks have hooks anyway.

Of course you can overload the built in functionality as this is currently not protected, but we might do that as well once MkIV is stable enough. After all, at the time of this writing overloading can be handy when testing.

This leaves the node list manipulators. The are grouped as follows:

category	**callback**	**usage**
processors	pre_linebreak_filter	called just before the paragraph is broken into lines
	hpack_filter	called just before a horizontal box is constructed
finalizers	post_linebreak_filter	called just after the paragraph has been broken into lines
shipouts	no callback yet	applied to the box (or xform) that is to be shipped out
mvlbuilders	buildpage_filter	called after some material has been added to the main vertical list
vboxbuilders	vpack_filter	called when some material is added to a vertical box

math	mlist_to_hlist	called just after the math list is created, before it is turned into an horizontal list

Each category has several subcategories but for users only two make sense: before and after. Say that you want to hook some tracing into the mvlbuilder. This is how it's done:

```
function third.mymodule.myfunction(where)
    nodes.show_simple_list(tex.lists.contrib_head)
end

nodes.tasks.appendaction("processors", "before", "third.mymodule.myfunction")
```

As you can see, in this case the function gets no head passed (at least not currently). This example also assumes that you know how to access the right items. The arguments and return values are given below.[9]

category	arguments	return value
processors	head, ...	head, done
finalizers	head, ...	head, done
shipouts	head	head, done
mvlbuilders		done
vboxbuilders	head, ...	head, done
math	head, ...	head, done

9.3 Tasks

In the previous section we already saw that the actions are in fact tasks and that we can append (and therefore also prepend) to a list of tasks. The before and after task lists are valid hooks for users contrary to the other tasks that can make up an action. However, the task builder is generic enough for users to be used for individual tasks that are plugged into the user hooks.

Of course at some point, too many nested tasks bring a performance penalty with them. At the end of a run MkIV reports some statistics and timings and these can give you an idea how much time is spent in Lua. Of course this is a rough estimate only.

[9] This interface might change a bit in future versions of ConTeXt. Therefore we will not discuss the few more optional arguments that are possible.

The following tables list all the registered tasks for the processors actions:

category	function
before	nodes.properties.attach
normalizers	typesetters.wrappers.handler
	typesetters.characters.handler
	fonts.collections.process
	fonts.checkers.missing
characters	scripts.autofontfeature.handler
	scripts.splitters.handler
	typesetters.cleaners.handler
	typesetters.directions.handler
	typesetters.cases.handler
	typesetters.breakpoints.handler
	scripts.injectors.handler
words	languages.replacements.handler
	languages.hyphenators.handler
	languages.words.check
	typesetters.initials.handler
	typesetters.firstlines.handler
fonts	builders.paragraphs.solutions.splitters.split
	nodes.handlers.characters
	nodes.injections.handler
	nodes.handlers.protectglyphs
	builders.kernel.ligaturing
	builders.kernel.kerning
	nodes.handlers.stripping
	fonts.goodies.colorschemes.coloring
lists	typesetters.characteralign.handler
	typesetters.spacings.handler
	typesetters.kerns.handler
	typesetters.digits.handler
	typesetters.italics.handler
	languages.visualizediscretionaries
after	typesetters.marksuspects

Some of these do have subtasks and some of these even more, so you can imagine that quite some action is going on there.

The finalizer tasks are:

category	function
before	unset
normalizers	unset
fonts	builders.paragraphs.solutions.splitters.optimize
lists	typesetters.paragraphs.normalize typesetters.margins.localhandler builders.paragraphs.keeptogether nodes.linefillers.handler
after	unset

Shipouts concern:

category	function
before	unset
normalizers	typesetters.showsuspects typesetters.margins.finalhandler builders.paragraphs.expansion.trace typesetters.alignments.handler nodes.references.handler nodes.destinations.handler nodes.rules.handler nodes.shifts.handler structures.tags.handler nodes.handlers.accessibility nodes.handlers.backgrounds nodes.handlers.alignbackgrounds nodes.properties.delayed
finishers	nodes.visualizers.handler attributes.colors.handler attributes.transparencies.handler attributes.colorintents.handler attributes.negatives.handler attributes.effects.handler attributes.viewerlayers.handler
after	unset

There are not that many mvlbuilder tasks currently:

category	function
before	unset

normalizers	streams.collect
	typesetters.margins.globalhandler
	nodes.handlers.migrate
	builders.vspacing.pagehandler
	builders.profiling.pagehandler
	typesetters.checkers.handler
after	unset

The vboxbuilder perform similar tasks:

category	**function**
before	unset
normalizers	builders.vspacing.vboxhandler
	typesetters.checkers.handler
after	unset

Finally, we have tasks related to the math list:

category	**function**
before	unset
normalizers	noads.handlers.showtree
	noads.handlers.unscript
	noads.handlers.variants
	noads.handlers.relocate
	noads.handlers.families
	noads.handlers.render
	noads.handlers.collapse
	noads.handlers.domains
	noads.handlers.autofences
	noads.handlers.resize
	noads.handlers.alternates
	noads.handlers.tags
	noads.handlers.italics
	noads.handlers.classes
builders	builders.kernel.mlist_to_hlist
	typesetters.directions.processmath
after	unset

As MkIV is developed in sync with LuaTEX and code changes from experimental to more final and reverse, you should not be too surprised if the registered

function names change.

You can create your own task list with:

```
nodes.tasks.new("mytasks",{ "one", "two" })
```

After that you can register functions. You can append as well as prepend them either or not at a specific position.

```
nodes.tasks.appendaction ("mytask","one","bla.alpha")
nodes.tasks.appendaction ("mytask","one","bla.beta")

nodes.tasks.prependaction("mytask","two","bla.gamma")
nodes.tasks.prependaction("mytask","two","bla.delta")

nodes.tasks.appendaction ("mytask","one","bla.whatever","bla.alpha")
```

Functions can also be removed:

```
nodes.tasks.removeaction("mytask","one","bla.whatever")
```

As removal is somewhat drastic, it is also possible to enable and disable functions. From the fact that with these two functions you don't specify a category (like `one` or `two`) you can conclude that the function names need to be unique within the task list or else all with the same name within this task will be disabled.

```
nodes.tasks.enableaction ("mytask","bla.whatever")
nodes.tasks.disableaction("mytask","bla.whatever")
```

The same can be done with a complete category:

```
nodes.tasks.enablegroup ("mytask","one")
nodes.tasks.disablegroup("mytask","one")
```

There is one function left:

```
nodes.tasks.actions("mytask",2)
```

This function returns a function that when called will perform the tasks. In this case the function takes two extra arguments in addition to `head`.[10]

[10] Specifying this number permits for some optimization but is not really needed

Tasks themselves are implemented on top of sequences but we won't discuss them here.

9.4 Paragraph and page builders

Building paragraphs and pages is implemented differently and has no user hooks. There is a mechanism for plugins but the interface is quite experimental.

10 Building paragraphs

10.1 Introduction

You enter the den of the Lion when you start messing around with the par-builder. Actually, as TEX does a pretty good job on breaking paragraphs into lines I never really looked into the code that does it all. However, the Oriental TEX project kind of forced it upon me. In the chapter about font goodies an optimizer is described that works per line. This method is somewhat similar to expansion level one support (hz) in the sense that it acts independent of the par builder: the split off (best) lines are postprocessed. Where expansion involves horizontal scaling, the goodies approach does with (Arabic) words what the original HZ approach does with glyphs.

It would be quite some challenge (at least for me) to come up with solutions that look at the whole paragraph and as the per-line approach works quite well, there is no real need for an alternative. However, in September 2008, when we were exploring solutions for Arabic par building, Taco converted the parbuilder into LUA code and stripped away all code related to hyphenation, protrusion, expansion, last line fitting, and some more. As we had enough on our plate at that time, we never came to really testing it. There was even less reason to explore this route because in the Oriental TEX project we decided to follow the "use advanced OPENTYPE features" route which in turn lead to the 'replace words in lines by narrower of wider variants' approach.

However, as the code was laying around and as we want to explore further I decided to pick up the parbuilder thread. In this chapter some experiences will be discussed. The following story is as much Taco's as mine.

10.2 Cleaning up

In retrospect, we should not have been too surprised that the first approximation was broken in many places, and for good reason. The first version of the code was a conversion of the C code that in turn was a conversion from the original interwoven PASCAL code. That first conversion still looked quite C–ish and carried interesting bit and pieces of C–macros, C–like pointer tests, interesting magic constants and more.

When I took the code and LUA-fied it nearly every line was changed and it took Taco and me a bit of reverse engineering to sort out all problems (thank you Skype). Why was it not an easy task? There are good reasons for this.

- The parbuilder (and related hpacking) code is derived from traditional TEX and has bits of pdfTEX, Aleph (Omega), and of course LuaTEX.

- The advocated approach to extending TEX has been to use change files which means that a coder does not see the whole picture.

- Originally the code is programmed in the literate way which means that the resulting functions are build stepwise. However, the final functions can (and have) become quite large. Because LuaTEX uses the woven (merged) code indeed we have large functions. Of course this relates to the fact that succesive TEX engines have added functionality. Eventually the source will be webbed again, but in a more sequential way.

- This is normally no big deal, but the Aleph (Omega) code has added a level of complexity due to directional processing and additional begin and end related boxes.

- Also the ε-TEX extension that deals with last line fitting is interwoven and uses goto's for the control flow. Fortunately the extensions are driven by parameters which make the related code sections easy to recognize.

- The pdfTEX protrusion extension adds code to glyph handling and discretionary handling. The expansion feature does that too and in addition also messes around with kerns. Extra parameters are introduced (and adapted) that influence the decisions for breaking lines. There is also code originating in pdfTEX which deals with poor mans grid snapping although that is quite isolated and not interwoven.

- Because it uses a slightly different way to deal with hyphenation, LuaTEX itself also adds some code.

- Tracing is sort of interwoven in the code. As it uses goto's to share code instead of functions, one needs to keep a good eye on what gets skipped or not.

I'm pretty sure that the code that we started with looks quite different from the original TEX code if it had been translated into C. Actually in modern TEX compiling involves a translation into C first but the intermediate form is not meant for human eyes. As the LuaTEX project started from that merged code, Taco and Hartmut already spent quite some time on making it more readable. Of course the original comments are still there.

Cleaning up such code takes a while. Because both languages are similar but also quite different it took some time to get compatible output. Because the C

code uses macros, careful checking was needed. Of course LUA's table model and local variables brought some work as well. And still the code looks a bit C–ish. We could not divert too much from the original model simply because it's well documented.

When moving around code redundant tests and orphan code has been removed. Future versions (or variants) might as well look much different as I want more hooks, clearly split stages, and convert some linked list based mechanism to LUA tables. On the other hand, as already much code has been written for CONTEXT MKIV, making it all reasonable fast was no big deal.

10.3 Expansion

The original C–code related to protrusion and expansion is not that efficient as many (redundant) function calls take place in the linebreaker and packer. As most work related to fonts is done in the backend, we can simply stick to width calculations here. Also, it is no problem at all that we use floating point calculations (as LUA has only floats). The final result will look okay as the original hpack routine will nicely compensate for rounding errors as it will normally distribute the content well enough. We are currently compatible with the regular par builder and protrusion code, but expansion gives different results (actually not worse).

The LUA hpacker follows a different approach. And let's admit it: most TEXies won't see the difference anyway. As long as we're cross platform compatible it's fine.

It is a well known fact that character expansion slows down the parbuilder. There are good reasons for this in the PDFTEX approach. Each glyph and intercharacter kern is checked a few times for stretch or shrink using a function call. Also each font reference is checked. This is a side effect of the way PDFTEX backend works as there each variant has its own font. However, in LUATEX, we scale inline and therefore don't really need the fonts. Even better, we can get rid of all that testing and only need to pass the eventual `expansion_ratio` so that the backend can do the right scaling. We will prototype this in the LUA version[11] and we feel confident about this approach it will be backported into the C code base. So eventually the C might become a bit more readable and efficient.

Intercharacter kerning is dealt with in a somewhat strange way. If a kern of

[11] For this Hartmuts has adapted the backend code has to honour this field in the glyph and kern nodes.

subtype zero is seen, and if it's neighbours are glyphs from the same font, the kern gets replaced by a scaled one looked up in the font's kerning table. In the parbuilder no real replacement takes place but as each line ends up in the hpack routine (where all work is simply duplicated and done again) it really gets replaced there. When discussing the current aproach we decided, that manipulating intercharacter kerns while leaving regular spacing untouched, is not really a good idea so there will be an extra level of configuration added to LuaTeX:[12]

0 no character and kern expansion
1 character and kern expansion applied to complete lines
2 character and kern expansion as part of the par builder
3 only character expansion as part of the par builder (new)

You might wonder what happens when you unbox such a list: the original font references have been replaced as were the kerns. However, when repackaged again, the kerns are replaced again. In traditional TeX, indeed rekerning might happen when a paragraph is repackaged (as different hyphenation points might be chosen and ligature rebuilding etc. has taken place) but in LuaTeX we have clearly separated stages. An interesting side effect of the conversion is that we really have to wonder what certain code does and if it's still needed.

10.4 Performance

We had already noticed that the Lua variant was not that slow. So after the first cleanup it was time to do some tests. We used our regular `tufte.tex` test file. This happens to be a worst case example because each broken line ends with a comma or hyphen and these will hang into the margin when protruding is enabled. So the solution space is rather large (an example will be shown later).

Here are some timings of the March 26, 2010 version. The test is typeset in a box so no shipout takes place. We're talking of 1000 typeset paragraphs. The times are in seconds an between parentheses the speed relative to the regular parbuilder is mentioned.

	native	**lua**	**lua + hpack**
normal	1.6	8.4 (5.3)	9.8 (6.1)
protruding	1.7	14.2 (8.4)	15.6 (9.2)

[12] As I more and more run into books typeset (not by TeX) with a combination of character expansion and additional intercharacter kerning I've been seriously thinking of removing support for expansion from ConTeXt MkIV. Not all is progress especially if it can be abused.

expansion	2.3	11.4 (5.0)	13.3 (5.8)
both	2.9	19.1 (6.6)	21.5 (7.4)

For a regular paragraph the LUA variant (currently) is 5 times slower and about 6 times when we use the LUA hpacker, which is not that bad given that it's interpreted code and that each access to a field in a node involves a function call. Actually, we can make a dedicated hpacker as some code can be omitted, The reason why the protruding is relatively slow is, that we have quite some protruding characters in the test text (many commas and potential hyphens) and therefore we have quite some lookups and calculations. In the C variant much of that is inlined by macros.

Will things get faster? I'm sure that I can boost the protrusion code and probably the rest as well but it will always be slower than the built in function. This is no problem as we will only use the LUA variant for experiments and special purposes. For that reason more MKIV like tracing will be added (some is already present) and more hooks will be provided once the builder is more compartimized. Also, future versions of LUATEX will pass around paragrapgh related parameters differently so that will have impact on the code as well.

10.5 Usage

The basic parbuilder is enabled and disabled as follows:[13]

```
\definefontfeature[example][default][protrusion=pure]
\definedfont[Serif*example]
\setupalign[hanging]

\startparbuilder[basic]
    \startcolor[blue]
        \input tufte
    \stopcolor
\stopparbuilder
```

This results in:

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,

[13] I'm not sure yet if the parbuilder has to do automatic grouping.

filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

There are a few tracing options in the `parbuilders` namespace but these are not stable yet.

10.6 Conclusion

The module started working quite well around the time that Peter Gabriels "Scratch My Back" ended up in my Squeezecenter: modern classical interpretations of some of his favourite songs. I must admit that I scratched the back of my head a couple of times when looking at the code below. It made me realize that a new implementation of a known problem indeed can come out quite different but at the same time has much in common. As with music it's a matter of taste which variant a user likes most.

At the time of this writing there is still work to be done. For instance the large functions need to be broken into smaller steps. And of course more testing is needed.

11 Tagged PDF

11.1 Introduction

Occasionally users asked me if ConTeXt can produce tagged pdf and the answer to that has been: I'll implement it when I need it. However, users tell me that publishers more and more demand tagged pdf files, although one might wonder what for, except maybe for accessibility. Another reason for not having spent too much time on it before is that the specification was not that inviting.

At any rate, when I saw Ross Moore[14] presenting tagged math at TUG 2010, I decided to look up the spec once more and see if I could get into the mood to implement tagging. Before I started it was already clear that there were a couple of boundary conditions:

- Tagging should not put a burden on the user but users should be able to tag themselves.
- Tagging should not slow down a run too much; this is no big deal as one can postpone tagging till the last run.
- Tagging should in no way interfere with typesetting, so no funny nodes should be injected.
- Tagging should not make the code look worse, neither the document source, nor the low level ConTeXt code.

And of course implementing it should not take more than a few days' work, certainly not in an exceptionally hot summer.

You can 'google' for one of Ross's documents (like `DML_002-2009-1_12.pdf`) to see how a document source looks at his end using a special version of pdfTeX. However, the version on my machine didn't support the shown primitives, so I could not see what was happening under the hood. Unfortunately it is quite hard to find a properly tagged document so we have only the reference manual as starting point. As the pdfTeX approach didn't look that pleasing anyway, I just started from scratch.

Tags can help Acrobat Reader when reading out the text aloud. But you cannot browse the structure in the no-cost version of Acrobat and as not all users have the professional version of Acrobat, the fact that a document has structure can go unnoticed. Add to that the fact that the overhead in terms of bytes is quite large as many more objects are generated, and you will understand why this

[14] He is often exploring the boundaries of pdf, Unicode and evolving techniques related to math publishing so you'd best not miss his presentations when you are around.

feature is not enabled by default.

11.2 Implementation

So, what does tagging boil down to? We can best look at how tagged information is shown in Acrobat. Figure 11.1 shows the content tree that has been added (automatically) to a document while figure 11.2 shows a different view.

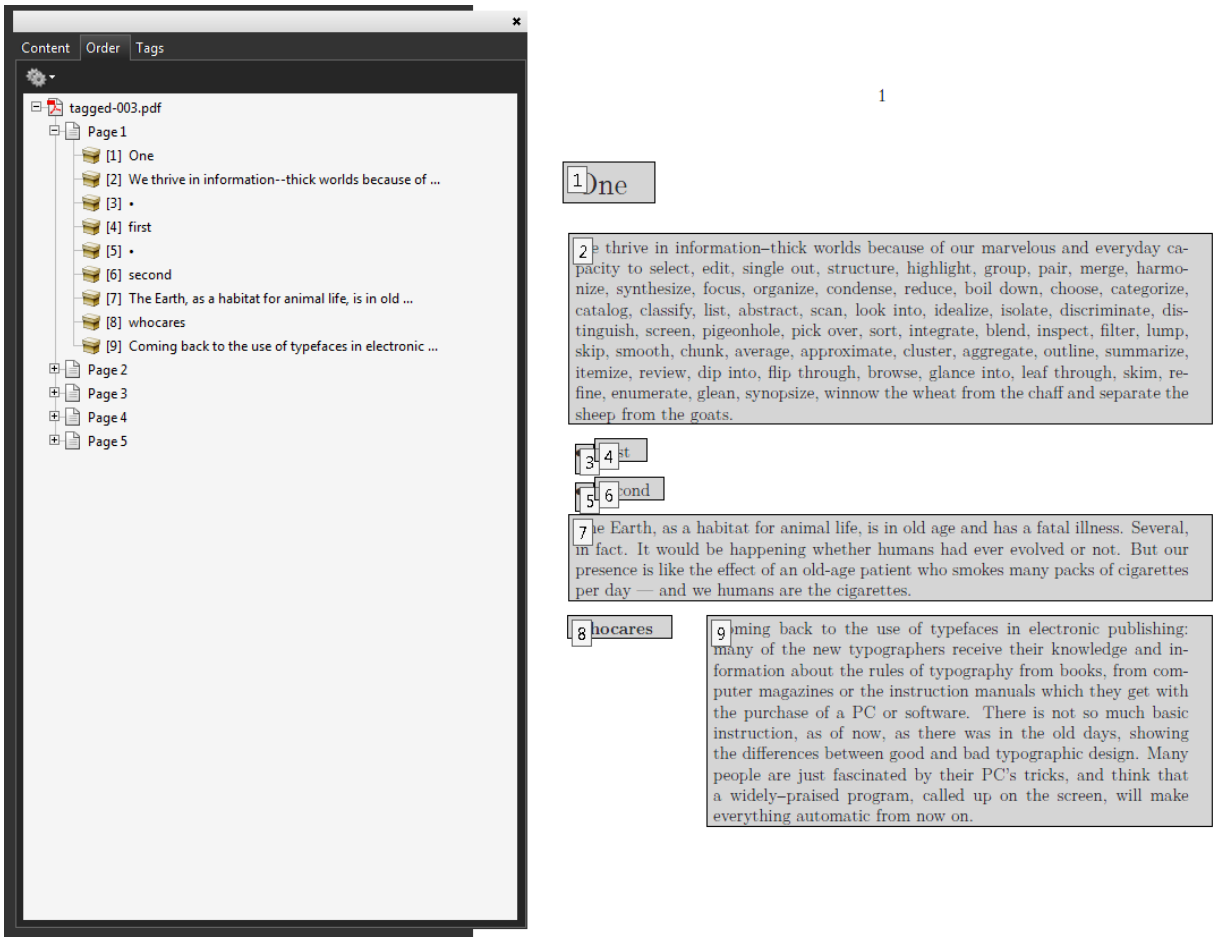


Figure 11.2 Acrobat showing the tag order.

In order to get that far, we have to do the following:

- Carry information with (typeset) text.
- Analyse this information when shipping out pages.
- Add a structure tree to the page.
- Add relevant information to the document.

That first activity is rather independent of the other three and we can use that information for other purposes as well, like identifying where we are in the document. We carry the information around using attributes. The last

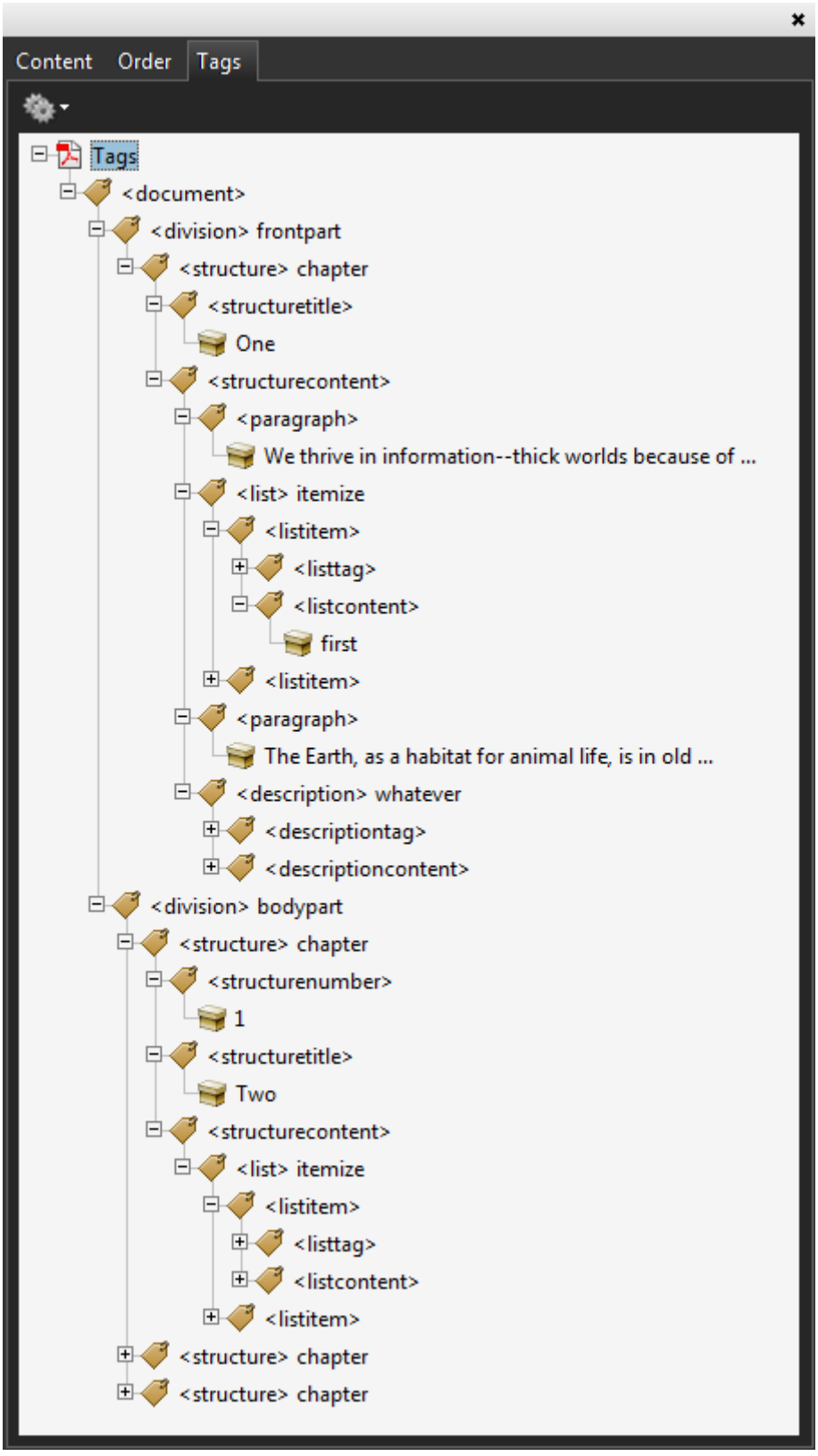


Figure 11.1 A tag list in Acrobat.

three activities took a bit of experimenting mostly using the “Example of Logical Structure” from the PDF standard 32000-1:2008.

This resulted in a tagging framework that uses explicit tags, meaning the user is responsible for the tagging:

```
\setupstructure[state=start,method=none]

\starttext

\startelement[document]

    \startelement[chapter]
        \startelement[p] \input davis \stopelement \par
    \stopelement

    \startelement[chapter]
        \startelement[p] \input zapf \stopelement \par
        \startelement[whatever]
            \startelement[p] \input tufte \stopelement \par
            \startelement[p] \input knuth \stopelement \par
        \stopelement
    \stopelement

    \startelement[chapter]
        oeps
        \startelement[p] \input ward \stopelement \par
    \stopelement

\stopelement

\stoptext
```

Since this is not much fun, we also provide an automated variant. In the previous example we explicitly turned off automated tagging by setting `method` to `none`. By default it has the value `auto`.

```
\setupstructure[state=start] % default is method=auto

\definedescription[whatever]

\starttext

\startfrontmatter
```

```
    \startchapter[title=One]
        \startparagraph \input tufte \stopparagraph
        \startitemize
            \startitem first \stopitem
            \startitem second \stopitem
        \stopitemize
        \startparagraph \input ward \stopparagraph
        \startwhatever {Herman Zapf} \input zapf \stopwhatever
    \stopchapter

\stopfrontmatter

\startbodymatter
    ..................
```

If you use commands like `\chapter` you will not get the desired results. Of course these can be supported but there is no real reason for it, as in MkIV we advise using the `start-stop` variant.

It will be clear that this kind of automated tagging brings with it a couple of extra commands deep down in ConTeXt and there (of course) we use symbolic names for tags, so that one can overload the built-in mapping.

```
\setuptaglabeltext[en][document=text]
```

As with other features inspired by viewer functionality, the implementation of tagging is independent of the backend. For instance, we can tag a document and access the tagging information at the TeX end. The backend driver code maps tags to relevant PDF constructs. First of all, we just map the tags used at the ConTeXt end onto themselves. But, as validators expect certain names, we use the PDF rolemap feature to map them to (less interesting) names. The next list shows the currently used internal names, with the PDF ones between parentheses.

combination (Span), combinationcaption (Span), combinationcontent (Span), combinationpair (Span), construct (Span), delimited (Quote), delimitedblock (BlockQuote), delimitedcontent (Span), delimitedsymbol (Span), description (Div), descriptioncontent (Div), descriptionsymbol (Span), descriptiontag (Div), division (Div), document (Div), float (Div), floatcaption (Caption), floatcontent (P), floatlabel (Span), floatnumber (Span), floattext (Span), formula (Div), formulacaption (Span), formulacontent (P), formulalabel (Span), formulanumber (Span), formulaset (Div), highlight (Span), ignore (Span), image (P), item (LI), itembody (Div), itemcontent (LBody), itemgroup (L),

itemhead (Div), itemtag (Lbl), label (Span), line (Code), lines (Code), link (Link), list (TOC), listcontent (P), listdata (P), listitem (TOCI), listpage (Reference), listtag (Lbl), maction (Span), margintext (Span), margintextblock (Span), math (Div), merror (Span), metadata (Div), metavariable (Span), mfenced (Span), mfrac (Span), mi (Span), mid (Span), mn (Span), mo (Span), mover (Span), mpgraphic (P), mroot (Span), mrow (Span), ms (Span), msqrt (Span), mstacker (Span), mstackerbot (Span), mstackermid (Span), mstackertop (Span), msub (Span), msubsup (Span), msup (Span), mtable (Table), mtd (TD), mtext (Span), mtr (TR), munder (Span), munderover (Span), number (Span), p (P), paragraph (P), private (Span), register (Div), registercontent (Span), registerentries (Div), registerentry (Div), registerlocation (Span), registerpage (Span), registerpagerange (Span), registerpages (Span), registersection (Div), registersee (Span), registerseparator (Span), registertag (Span), section (Sect), sectioncontent (Div), sectionnumber (H), sectiontitle (H), sorting (Span), sub (Span), subformula (Div), subsentence (Span), subsentencecontent (Span), subsentencesymbol (Span), subsup (Span), sup (Span), synonym (Span), table (Table), tablecell (TD), tablerow (TR), tabulate (Table), tabulatecell (TD), tabulaterow (TR), verbatim (Code), verbatimblock (Code), verbatimline (Code), verbatimlines (Code).

So, the internal ones show up in the tag trees as shown in the examples but applications might use the rolemap which normally has less detail.

Because we keep track of where we are, we can also use that information for making decisions.

```
\doifinelementelse{structure:section}            {yes} {no}
\doifinelementelse{structure:chapter}            {yes} {no}
\doifinelementelse{division:*-structure:chapter} {yes} {no}
\doifinelementelse{division:*-structure:*}       {yes} {no}
```

As shown, you can use `*` as a wildcard. The elements are separated by `-`. If you don't know what tags are used, you can always enable the tag related tracker:

```
\enabletrackers[structure.tags]
```

This tracker reports the identified element chains to the console and log.

11.3 Special care

Of course there are a few complications. First of all the tagging model sort of contradicts the concept of a nicely typeset document where structure and outcome are not always related. Most TEX users are aware of the fact that TEX does

not have space characters and does a great job on kerning and hyphenation. The tagging machinery on the other hand uses a rather dumb model of strings separated by spaces.[15] But we can trick TEX into providing the right information to the backend so that words get nicely separated. The non-optimized function that does this looks as follows:

```
function injectspaces(head)
    local p
    for n in node.traverse(head) do
        local id = n.id
        if id == node.id("glue") then
            if p and p.id == node.id("glyph") then
                local g = node.copy(p)
                local s = node.copy(n.spec)
                g.char, n.spec = 32, s
                p.next, g.prev = g, p
                g.next, n.prev = n, g
                s.width = s.width - g.width
            end
        elseif id == node.id("hlist") or id == node.id("vlist") then
            injectspaces(n.list,attribute)
        end
        p = n
    end
end
```

Here we squeeze in a space (given that it is in the font which it normally is when you use ConTEXt) and make a compensation in the glue. Given that your page sits in box 255, you can do this just before shipping the page out:

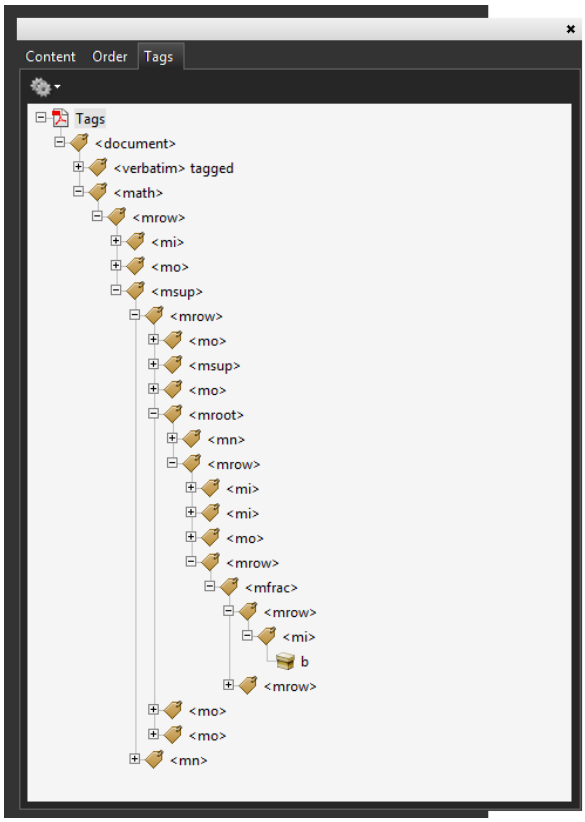
```
injectspaces(tex.box[255].list)
```

Then there are the so-called suspects: things on the page that are not related to structure at all. One is supposed to tag these specially so that the built-in reading equipment is not confused. So far we could get around them simply because they don't get tagged at all and therefore are not seen anyway. This might well be enough of a precaution.

Of course we need to deal with mathematics. Fortunately the presentation MathML model is rather close to TEX and so we can map onto that. After all we don't need to care too much about back-mapping here. The currently present code is rather experimental and might get extended or thrown out in favour of

[15] The search engine on the other hand is rather clever on recognizing words.

inline MathML. Figure 11.3 demonstrates that a first approach does not even look that bad. In future versions we might deal with table-like math constructs, like matrices.



1

```
$$ y = \left( x^2 + \root{3}\of{ax + {{b}\over{3}}} \right) ^ 2 $$
```

$$y = \left(x^2 + \sqrt[3]{ax + \frac{b}{3}} \right)^2$$

Figure 11.3 Experimental math tagging.

This is a typical case where more energy has to be spent on driving the voice of Acrobat but I will do that when we find a good reason.

As mentioned, it will take a while before all relevant constructs in ConTeXt support tagging, but support is already quite complete. Some screen dumps are included as examples at the end.

11.4 Conclusion

Surprisingly, implementing all this didn't take that much work. Of course detailed automated structure support from the complete ConTeXt kernel will take some time to get completed, but that will be done on demand and when we run into missing bits and pieces. It's still not decided to what extent alternate representations and alternate texts will be supported. Experiments with the reading-aloud machinery are not satisfying yet but maybe it just can't get any better. It would be nice if we could get some tags being announced without overloading the content, that is: without using ugly hacks.

And of course, code like this is never really finished if only because PDF evolves. Also, it is yet another nice test case and torture test for LuaTeX and it helps us to find buglets and oversights.

11.5 Some more examples

In ConTeXt we have user definable verbatim environments. As with other user definable environments we show the specific instance as comment next to the structure component. See figure 11.4. Some examples of tables are shown in figure 11.5. Future versions will have a bit more structure. Tables of contents (see figure 11.6) and registers (see figure 11.7) are also tagged. (One might wonder what the use is of this.) In figure 11.8 we see some examples of floats. External images as well as MetaPost graphics are tagged as such. This example also shows an example of a user environment, in this case:

```
\definestartstop[notabene][style=\bf]
```

In a similar fashion, footnotes (figure 11.9) end up in the structure tree, but in the typeset document they move around (normally forward when there is no room).

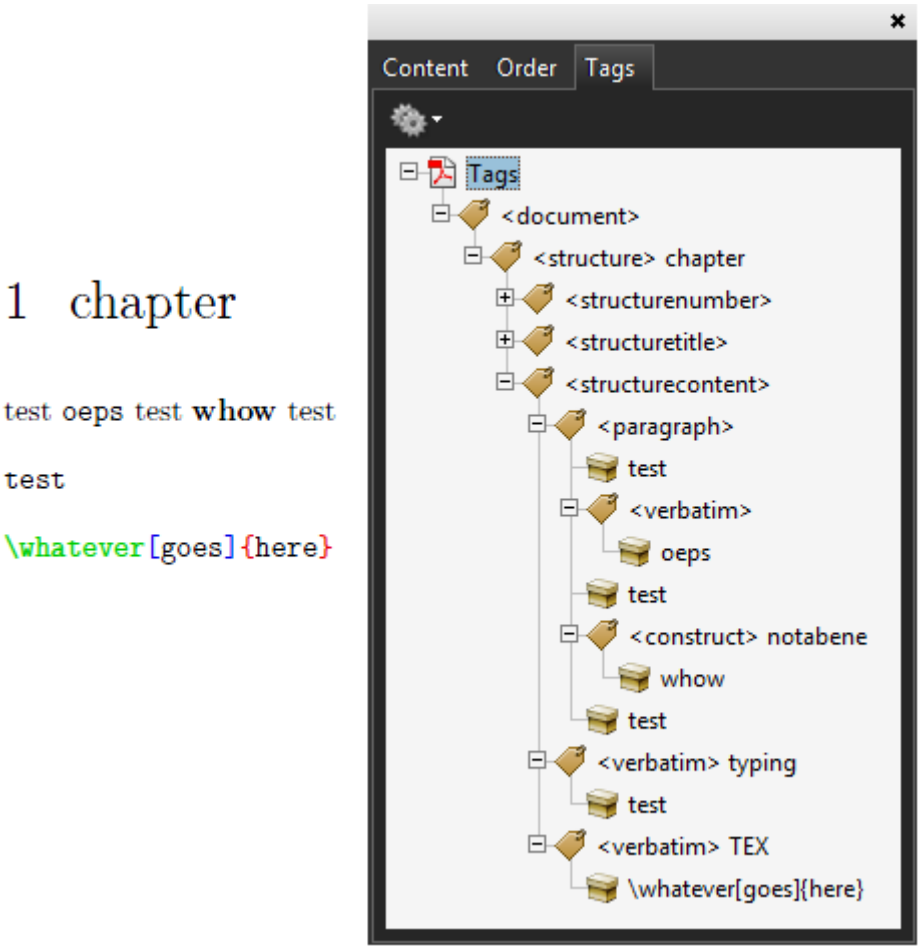


Figure 11.4 Verbatim, including dedicated instances.

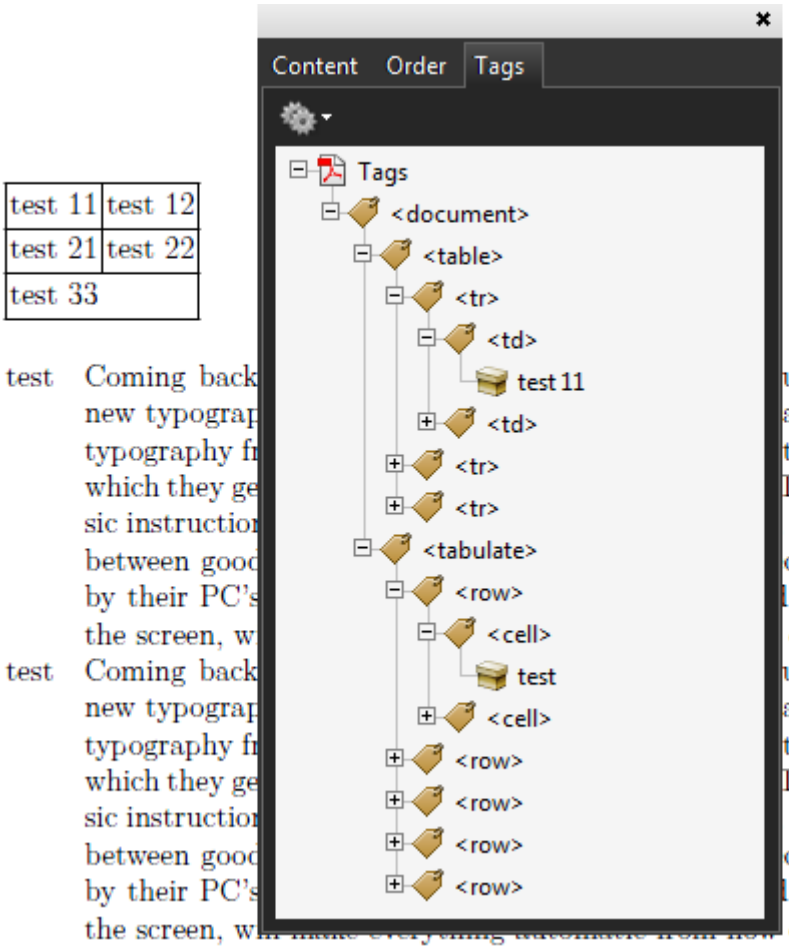


Figure 11.5 Natural tables as well as the tabulate mechanism is supported.

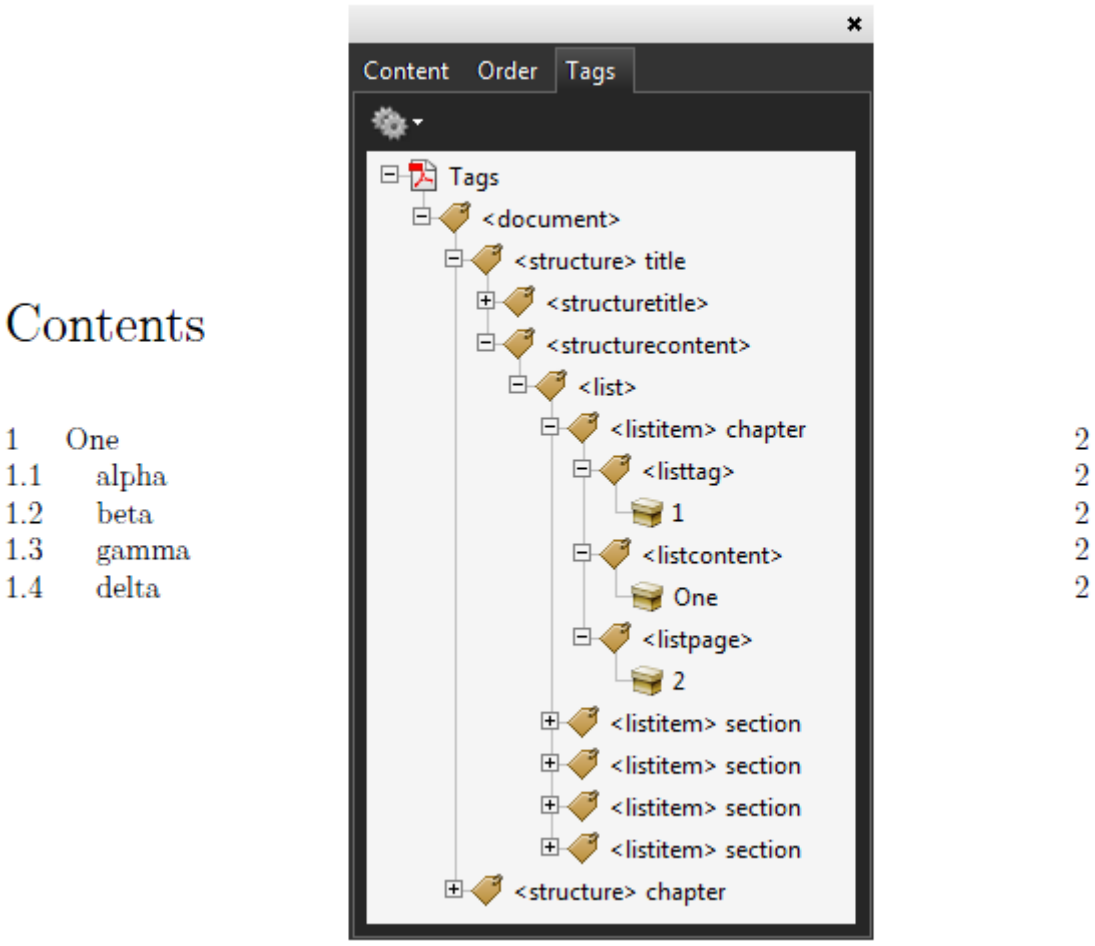


Figure 11.6 Tables of content with specific entries tagged.

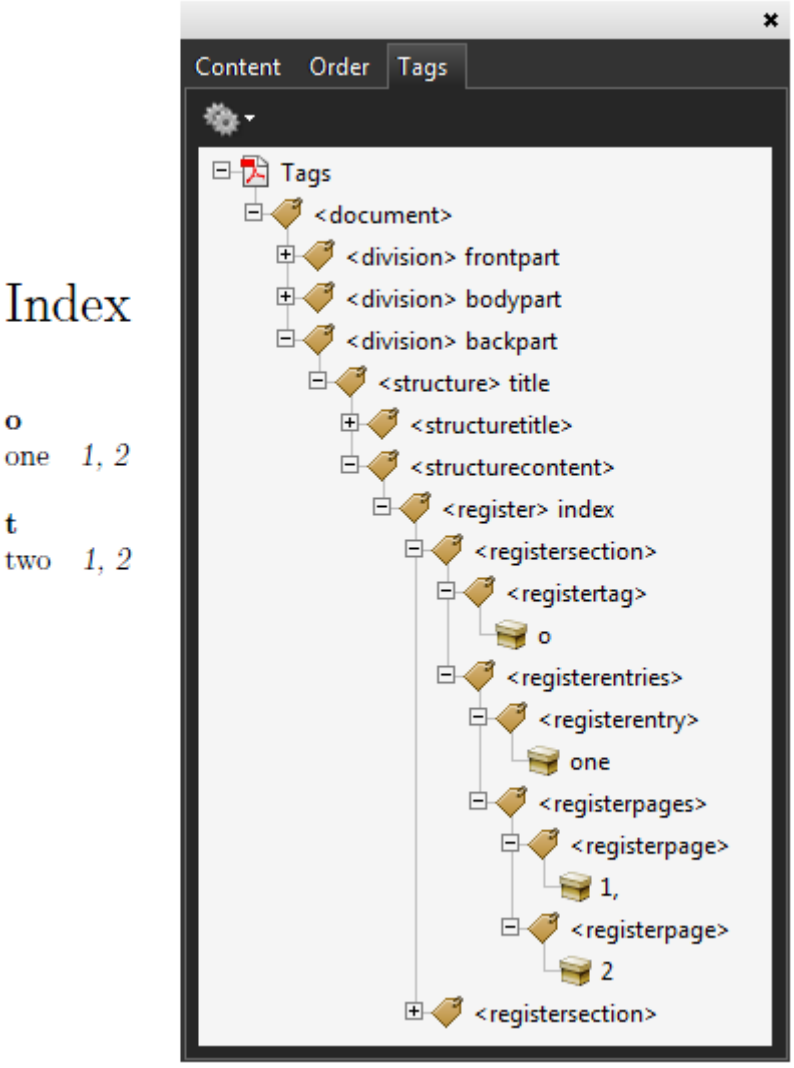


Figure 11.7 A detailed view of registered is provided.

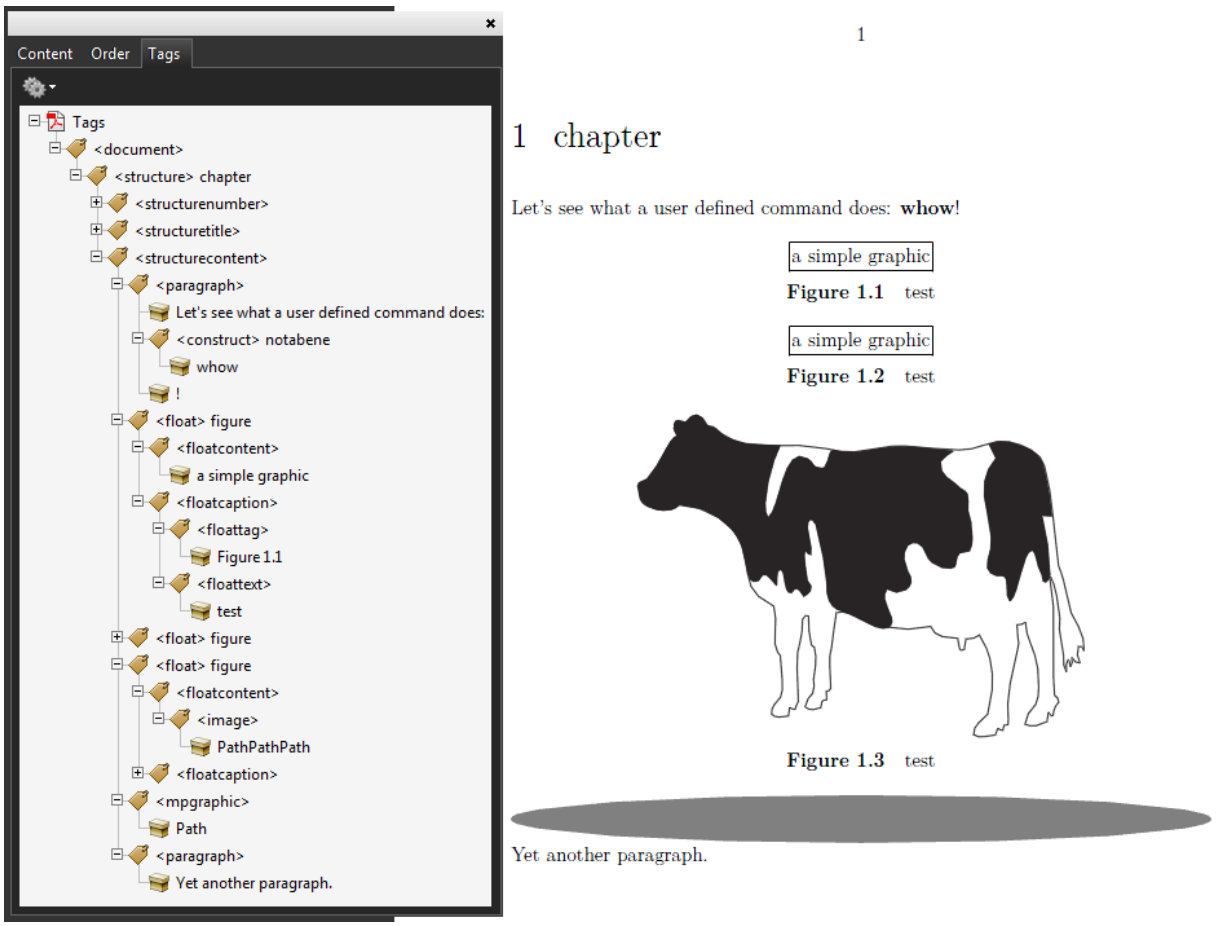


Figure 11.8 Floats tags end up in text stream. Watch the user defined construct.

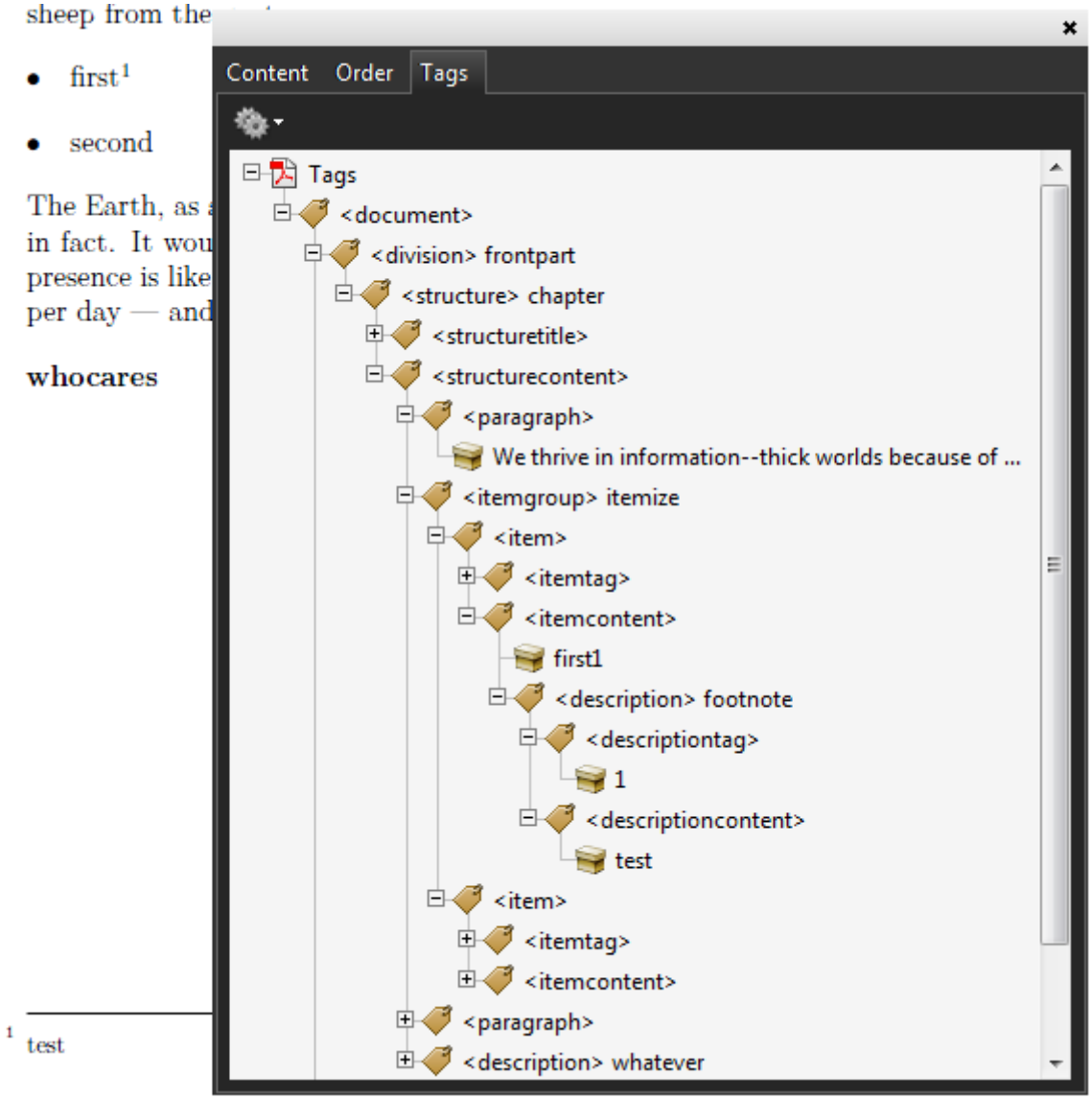


Figure 11.9 Footnotes are shown at the place in the input (flow).

12 Including pages

12.1 Introduction

It is tempting to add more and more features to the backend code of the engine but it is not really needed. Of course there are features that can best be supported natively, like including images. In order to include PDF images in LuaTeX the backend uses a library (xpdf or poppler) that can load an page from a file and embed that page into the final PDF, including all relevant (indirect) objects needed for rendering. In LuaTeX an experimental interface to this library is included, tagged as `epdf`. In this chapter I will spend a few words on my first attempt to use this new library.

12.2 The library

The interface is rather low level. I got the following example from Hartmut (who is responsible for the LuaTeX backend code and this library).

```
local doc = epdf.open("luatexref-t.pdf")
local cat = doc:getCatalog()
local pag = cat:getPage(3)
local box = pag:getMediaBox()

local w = pag:getMediaWidth()
local h = pag:getMediaHeight()
local n = cat:getNumPages()
local m = cat:readMetadata()

print("nofpages: ", n)
print("metadata: ", m)
print("pagesize: ", w .. " * " .. h)
print("mediabox: ", box.x1, box.x2, box.y1, box.y2)
```

As you see, there are accessors for each interesting property of the file. Of course such an interface needs to be extended when the PDF standard evolves. However, once we have access to the so called catalog, we can use regular accessors to the dictionaries, arrays and other data structures. So, in fact we don't need a full interface and can draw the line somewhere.

There are a couple of things that you normally do not want to deal with. A PDF file is in fact just a collection of objects that form a tree and each object can be reached by an index using a table that links the index to a position in

the file. You don't want to be bothered with that kind of housekeeping indeed. Some data in the file, like page objects and annotations are organized in a tree form that one does not want to access in that form, so again we have something that benefits from an interface. But the majority of the objects are simple dictionaries and arrays. Streams (these hold the document content, image data, etc.) are normally not of much interest, but the library provides an interface as you can bet on needing it someday. The library also provides ways to extend the loaded PDF file. I will not discuss that here.

Because in ConTeXt we already have the `lpdf` library for creating PDF structures, it makes sense to define a similar interface for accessing PDF. For that I wrote a wrapper that will be extended in due time (read: depending on needs). The previous code now looks as follows:

```
local doc = epdf.open("luatexref-t.pdf")
local cat = doc.Catalog
local pag = cat.Pages[3]
local box = pag.MediaBox

local llx, lly, urx, ury = box[1], box[2] box[3], box[4]

local w = urx - llx -- or: box.width
local h = ury - lly -- or: box.height
local n = cat.Pages.size
local m = cat.Metadata.stream

print("nofpages: ", n)
print("metadata: ", m)
print("pagesize: ", w .. " * " .. h)
print("mediabox: ", llx, lly, urx, ury)
```

If we write code this way we are less dependent on the exact API, especially because the `epdf` library uses methods to access the data and we cannot easily overload method names in there. When you look at the `box`, you will see that the natural way to access entries is using a number. As a bonus we also provide the `width` and `height` entries.

12.3 Merging links

It has always been on my agenda to add the possibility to carry the (link) annotations with an included page from a document. This is not that much needed in a regular document, but it can be handy when you use ConTeXt to assemble documents. In any case, such a merge has to happen in such a way that

it does not interfere with other links in the parent document. Supporting this in the engine is no option as each macro package follows its own approach to referencing and interactivity. Also, demands might differ and one would end up with a lot of (error prone) configurability. Of course we want scaled pages to behave well too.

Implementing the merge took about a day and most of that time was spent on experimenting with the `epdf` library and making the first version of the wrapper. I definitely had expected to waste more time on it. So, this is yet another example of extensions that are quite doable in the LuaTEX mix. Of course it helps that the ConTEXt graphic inclusion code provides enough information to integrate such a feature. The merge is controlled by the interaction key, as shown here:

```
\externalfigure[somefile.pdf][page=1,scale=700,interaction=yes]
\externalfigure[somefile.pdf][page=2,scale=600,interaction=yes]
\externalfigure[somefile.pdf][page=3,scale=500,interaction=yes]
```

You can finetune the merge by providing a list of options to the interaction key but that's still somewhat experimental. As a start the following links are supported.

- internal references by name (often structure related)
- internal references by page (e.g. table of contents)
- external references by file (optionally by name and page)
- references to uri's (normally used for webpages)

When users like this functionality (or when I really need it myself) more types of annotations can be added although support for JavaScript and widgets doesn't make much sense. On the other hand, support for destinations is currently somewhat simplified but at some point we will support the relevant zoom options.

The implementation is not that complex:

- check if the included page has annotations
- loop over the list of annotations and determine if an annotation is supported (currently links)
- analyze the annotation and overlay a button using the destination that belongs to the annotation

Now, the reason why we can keep the implementation so simple is that we just map onto existing ConTEXt functionality. And, as we have a rather integrated support for interactive actions, only a few basic commands are involved.

Although we could do that all in LUA, we delegate this to TEX. We create a layer which we put on top of the image. Links are put onto this layer using the equivalent of:

```
\setlayer
  [epdflinks]
  [x=...,y=...,preset=leftbottom]
  {\button
     [width=...,height=...,offset=overlay,frame=off]
     {}% no content
     [...]}}
```

The `\button` command is one of those interaction related commands that accepts any action related directive. In this first implementation we see the following destinations show up:

```
somelocation
url(http://www.pragma-ade.com)
file(somefile)
somefile::somelocation
somefile::page(10)
```

References to pages become named destinations and are later resolved to page destinations again, depending on the configuration of the main document. The links within an included file get their own namespace so (hopefully) they will not clash with other links.

We could use lower level code which is faster but we're not talking of time critical code here. At some point I might optimize the code a bit but for the moment this variant gives us some tracing options for free. Now, the nice thing about using this approach is that the already existing cross referencing mechanisms deal with the details. Each included page gets a unique reference so references to not included pages are ignored simply because they cannot be resolved. We can even consider overloading certain types of links or ignoring named destinations that match a specific pattern. Nothing is hard coded in the engine so we have complete freedom of doing that.

12.4 Merging layers

When including graphics from other applications it might be that they have their content organized in layers (that then can be turned on or off). So it will be no surprise that on the agenda is merging layer information: first a straightforward inclusion of optional content dictionaries, but it might make

sense to parse the content stream and replace references to layers by those that are relevant in the main document. Especially when graphics come from different sources and layer names are inconsistent some manipulation might be needed so maybe we need more detailed control. Implementing this is is no big deal and mostly a matter of figuring out a clean and simple user interface.

13 Exporting XML

13.1 Introduction

Every now and then on the the mailing list users ask if ConTeXt can produce HTML instead of for instance PDF, and the answer has always been unsatisfying. In this chapter I will present the MkIV way of doing this.

13.2 The clumsy way

My favourite answer to the question about how to produce HTML (or more general XML as it can be transformed) has always been: "I'd just typeset it!". Take:

```
\def\MyChapterCommand#1#2{<h1>#2</h1>}
\setuphead[chapter][command=\MyChapterCommand]
```

Here `\chapter{Hello World}` will produce:

```
<h1>Hello World</h1>
```

Now imagine that you hook such commands into all relevant environments and that you use a style with no header and footer lines. You use a large page (A2) and a small monospaced font (4pt) so that page breaks will not interfere too much. If you want columns, fine, just hook in some code that typesets the final columns as tables. In the end you will have an ugly looking PDF file but by feeding it into `pdftotext` you will get a nicely formatted HTML file.

For some languages of course encoding issues would show up and there can be all kind of interferences, so eventually the amount of code dealing with it would have accumulated. This is why we don't follow this route.

An alternative is to use `tex4ht` which does an impressive job for LaTeX, and supports ConTeXt to some extent as well. As far as I know it overloads some code deep down in the kernel which is something 'not done' in the ConTeXt universe if only because we cannot keep control over side effects. It also complicates maintainance of both systems.

In MkIV however, we do have the ability to export the document to a structured XML file so let's have a look at that.

13.3 Structure

The ability to export to some more verbose format depends on the availability of structural information. As we already tag elements for the sake of tagged PDF, it was tempting to see how well we could use those tags for exporting to XML. In principle it is possible to use Acrobat Professional to export the content using tags but you can imagine that we get a better quality if we stay within the scope of the producing machinery.

```
\setupbackend[export=yes]
```

This is all you need unless you want to fine tune the resulting XML file. If you are familiar with tagged PDF support in ConTeXt, you will recognize the result. When you process the following file:

```
\setupbackend[export=yes]

\starttext

\startchapter[title=Test]
A paragraph.\par Another paragraph.
\stopchapter

\stoptext
```

You will get a file with the suffix `export` that looks as follows:[16]

It's no big deal to postprocess such a file. In that case one can for instance ignore the chapter number or combine the number and the title. Of course rendering information is lost here. However, sometime it makes sense to export some more details. Take the following table:

```
\starttext

\bTABLE
    \bTR \bTD test 1.1 \eTD \bTD[ny=2] test 1.2 \eTD \eTR
    \bTR \bTD test 2.1 \eTD                     \eTR
    \bTR \bTD test 3.1 \eTD \bTD test 3.2       \eTD \eTR
    \bTR \bTD test 4.1 \eTD \bTD                \eTD \eTR
    \bTR \bTD[nx=2,align=flushright] test 5.1   \eTD \eTR
\eTABLE
```

[16] We will omit the topmost lines in following examples.

```
\stoptext
```

Here we need to preserve the span related information as well as cell specific alignments as for tables this is an essential part of the structure.

The tabulate mechanism is quite handy for regular text especially when the content of cells has to be split over pages. As each line in a paragraph in a tabulate becomes a cell, we need to reconstruct the paragraphs from the (split) alignment cells.

```
\starttext

\starttabulate[|l|p|r|]
    \NC zero  \NC line one \par line two \par line three \NC 0 \NC \NR
  % \NC one   \NC \input zapf \par \input zapf           \NC 1 \NC \NR
    \NC two   \NC before \type {connect} \par after     \NC 2 \NC \NR
    \NC three \NC before \type {connect} after          \NC 3 \NC \NR
    \NC four  \NC before \break inbetween \par after    \NC 4 \NC \NR
\stoptabulate

\stoptext
```

This becomes:

The `<break/>` elements are injected automatically between paragraphs. We could tag each paragraph individually but that does not work that well when we have for instance a quotation that spans multiple paragraphs (and maybe starts in the middle of one). An empty element is not sensitive for this and is still a signal that vertical spacing is supposed to be applied.

13.4 The implementation

We implement tagging using attributes. The advantage of this is that it does not interfere with typesetting, but a disadvantage is that not all parent elements are visible. When we encounter some content, we're in the innermost element so if we want to do something special, we need to deduce the structure from the current child. This is no big deal as we have that information available at each child element in the tree.

The first implementation just flushed the XML on the fly (i.e. when traversing the node list) but when I figured out that collapsing was needed for special cases like tabulated paragraphs this approach was no longer valid. So, after

some experiments I decided to build a complete structure tree in memory[17]. This permits us to handle situations like the following:

```
\starttext

\startitemize[n]
    \startitem one \stopitem
    \startitem two \stopitem
\stopitemize

\startitemize[packed,a]
    \startitem \quote{one} \stopitem
    \startitem \quote{two} \stopitem
\stopitemize

\stoptext
```

Here we get:

The `symbol` and `packed` attributes are first seen at the `itemcontent` level (the innermost element) so when we flush the `itemgroup` element's attributes we need to look at the child elements (content) that actually carry the attribute.[18]

I already mentioned collapsing. As paragraphs in a tabulate get split into cells, we encounter a mixture that cannot be flushed sequentially. However, as each cell is tagged uniquely we can append the lines within a cell. Also, as each paragraph gets a unique number, we can add breaks before a new paragraph starts. Collapsing and adding breakpoints is done at the end, and not per page, as paragraphs can cross pages. Again, thanks to the fact that we have a tree, we can investigate content and do this kind of manipulations.

Moving data like footnotes are somewhat special. When notes are put on the page (contrary to for instance end notes) the so called 'insert' mechanism is used where their content is kept with the line where it is defined. As a result we see them end up instream which is not that bad a coincidence. However, as in MkIV notes are built on top of (enumerated) descriptions, we need to distinguish them somehow so that we can cross reference them in the export.

```
\starttext

\startchapter[title=Notes]
```

[17] We will see if this tree will be used for other purposes in the future.
[18] Only glyph nodes are investigated for structure.

```
test \footnote[a]{note a}
test \footnote[b]{note b}

\stopchapter

\stoptext
```

Currently this will end up as follows:

Graphics are also tagged and the `image` element reflects the included image.

```
\starttext

\placefigure
    [here] [fig:cow]
    {It looks like a cow.}
    {\externalfigure[cow.pdf]}

\stoptext
```

If the image sits on another path then that path shows up in an attribute and when a page other than 1 is taken from the (pdf) image, it gets mentioned as well.

Cross references are another relevant aspect of an export. In due time we will export them all. It's not so much complicated because all information is there but we need to hook some code into the right spot and making examples for those cases takes a while as well.

```
\setupinteraction[state=start]

\starttext

\startchapter[title=One,reference=alpha]
    In \in{chapter}[beta] ...
\stopchapter

\startchapter[title=Two,reference=beta]
    In \in{chapter}[alpha] ...
\stopchapter

\stoptext
```

We export references in the ConTeXt specific way, so no interpretation takes place.

As ConTeXt has an integrated referencing system that deals with internal as well as external references, url's, special interactive actions like controlling widgets and navigations, etc. and we export the raw reference specification as well as additional attributes that provide some detail.

```
\setupinteraction[state=start]

\useurl [pragma] [www.pragma-ade.com]

\starttext

\startparagraph
    You can visit \goto{pragma}[url(www.pragma-ade.com)].
\stopparagraph

\startparagraph
    You can visit \goto{pragma}[url(pragma)].
\stopparagraph

\stoptext
```

Of course, when postprocessing the exported data, you need to take these variants into account.

13.5 Math

Of course there are limitations. For instance TeXies doing math might wonder if we can export formulas. To some extent the export works quite well.

```
\starttext

Is it $ e = mc^2 $ maybe:

\startformula
m = \frac{\sqrt{e}}{c}
\stopformula

\stoptext
```

This results in the usual rather verbose presentation MathML:

More complex math (like matrices) will be dealt with in due time as for this Aditya and I have to take tagging into account when we revisit the relevant code as part of the MkIV cleanup and extensions. It's not that complex but it makes no sense to come up with intermediate solutions.

Display verbatim is also supported. In this case we tag individual lines.

```
\starttext

\starttyping
line one
line two
\stoptyping

\stoptext
```

The export is not that spectacular:

A rather special case are marginal notes. We do tag them because they often contain usefull information.

```
\starttext

\startparagraph
    test \inleft{left 1} test
\stopparagraph

\margintitle{left 2}

\startparagraph
    test test
\stopparagraph

\startparagraph
    \inrightmargin{\slanted{right 1}}test
\stopparagraph

\stoptext
```

The output is currently as follows:

However, this might change in future versions.

13.6 Formatting

The output is formatted using indentation and newlines. The extra run time needed for this (actually, quite some of the code is related to this) is compensated by the fact that inspecting the result becomes more convenient. Each environment has one of the properties `inline`, `mixed` and `display`. A display environment gets newlines around it and an inline environment none at all. The mixed variant does something in between. In the following example we tag some user elements, but you can as well influence the built in ones.

```
\setelementnature[display][display]
\setelementnature[inline] [inline]
\setelementnature[mixed]  [mixed]

\starttext

\startelement[display]
    \startelement[inline]
        test
        \startelement[display]
            test
        \stopelement
        test
    \stopelement
\stopelement

\stoptext
```

This results in:

Keep in mind that elements have no influence on the typeset result apart from introducing spaces when used this way (this is not different from other TEX commands). In due time the formatting might improve a bit but at least we have less chance ending up with those megabyte long one–liners that some applications produce.

13.7 A word of advise

In (for instance) HTML class attributes are used to control rendering driven by stylesheets. In CONTEXT you can often define derived environments and their names will show up in the detail attribute. So, if you want control at that level in the export, you'd better use the structure related options built in CONTEXT, for instance:

```
\definehead[specialsection][section]

\starttext

\startsection[title=Normal section]
    normal
\stopsection

\startspecialsection[title=Special section]
    special
\stopspecialsection

\stoptext
```

This gives two different sections:

13.8 Conclusion

It is an open question if such an export is useful. Personally I never needed a feature like this and there are several reasons for this. First of all, most of my work involves going from (often complex) XML to PDF and if you have XML as input, you can also produce HTML from it. For documents that relate to ConTeXt I don't need it either because manuals are somewhat special in the sense that they often depend on showing something that ends up on paper (or its screen counterpart) anyway. Loosing the makeup also renders the content somewhat obsolete. But this feature is still a nice proof of concept anyway.

14 Optimizations again

14.1 Introduction

Occasionally we do some timing on new functionality in either LuaTeX or MkIV, so here's another wrapup.

14.2 Font loading

In ConTeXt we cache font data in a certain way. Loading a font from the cache takes hardly any time. However, preparation takes more time as well memory as we need to go from the fontforge ordering to one we can use. In MkIV we have several font tables:

- The original fontforge table: this one is only loaded once and converted to another representation that is cached.
- The cached font representation that is the basis for further manipulations.
- In base mode this table is converted to a (optionally cached) scaled tfm table that is passed to TeX.
- In node mode a limited scaled version is passed to TeX. As with base mode, this table is kept in memory so that we can access the data.
- When processing features in node mode additional (shared) subtables are created that extend the memorized catched table.

This model is already quite old and dates from the beginning of MkIV. Future versions might use different derived tables but for the moment we need all this data if only because it helps us with the development.

The regular method to construct a font suitable for TeX, either or not using base mode or node mode in MkIV, is to load the font as table using `to_table`, a `fontloader` method. This means that all information is available (and can be manipulated). In MkIV this table is converted to another one and in the process new entries are added and existing ones are freed. Quite some garbage collection and table resizing takes place in the process. In the cached instance we share identical tables so there we can gain a lot of memory and avoid garbage collection.

The difference in usage is as follows:

```
do
  local f = fontloader.open("somefont.otf") -- allocates font object
  local t = fontloader.to_table(f)          -- allocates table
```

```
  fontloader.close(f)                          -- frees font object
  for index, glyph in pairs(t) do
    local width = glyph.width                  -- accesses table value
  end
end                                            -- frees table
```

Here `t` is a complete LUA table and it can get quite large: script fonts like Zapfino (for latin) or Husayni (for arabic) have lots of alternate shapes and much features related information, fonts meant for CJK usage have tens of thousands of glyphs, and math fonts like Cambria have many glyphs and math specific information.

```
do
  local f = fontloader.open("somefont.otf") -- allocates font object
  for index=0, t.glyphmax-1 do
    local glyph = f.glyphs[index]            -- assigns user data object
    if glyph then
      local width = glyph.width              -- calls virtual table value
    end
  end
  fontloader.close(f)                          -- frees font object
end
```

In this case there is no big table, and `glyph` is a so called userdata object. Its entries are created when asked for. So, where in the first example the `width` of a glyph is a number, in the second case it is a function disguised as virtual key that will return a number. In the first case you can change the width, in the second case you can't.

This means that if you want to keep the data around you need to copy it into another table but you can do that stepwise and selectively. Alternatively you can keep the font object in memory. As some glyphs can have much data you can imagine that when you only need to access the width, the userdata method is more efficient. On the other hand, if you need access to all information, the first method is more interesting as less overhead is involved.

In the userdata variant only the parent table and its glyph subtable are virtualized, as are entries in an optional subfonts table. So, if you ask for the kerns table of a glyph you will get a real table as it makes no sense to virtualize it. A way in between would have been to request tabls per glyph but as we will see there is no real benefit in that while it would further complicate the code.

When in LUATEX 0.63 the loaded font object became partially virtual it was time

to revision the loading code to see if we could benefit from this.

In the following tables we distinguish three cases: the original but adapted loading code[19], already a few years old, the new sparse loading code, using the userdata approach and no longer a raw table, and a mixed approach where we still use the raw table but instead of manipulating that one, construct a new one from it. It must be noticed that in the process of integrating the new method the traditional method suffered.

First we tested Oriental TEX's Husayni font. This one has lots of features, many of lookups, and quite some glyphs. Keep in mind that the times concern the preparation and not the reload from the cache, which is more of less neglectable. The memory consumption is a snapshot of the current run just after the font has been loaded. Peak memory is what bothers most users. Later we will explain what the values between parenthesis refer to.

	used memory	**peak memory**	**font loading time**
table	113 MB (102)	118 MB (117)	1.8 sec (1.9)
mixed	114 MB (103)	119 MB (117)	1.9 sec (1.9)
sparse	117 MB (104)	121 MB (120)	1.9 sec (2.0)
cached	75 MB	80 MB	0.4 sec
baseline	67 MB	71 MB	0.3 sec

So, here the new method is not offering any advantages. As this is a font we use quite a lot during development, any loading variant will do the job with similar efficiency.

Next comes Cambria, a font that carries lots of glyphs and has extensive support for math. In order to provide a complete bodyfont setup some six instances are loaded. Interesting is that the original module needs 3.9 seconds instead if 6.4 which is probably due to a different ordering of code which might influence the garbage collector and it looks like in the reorganized code the garbage collector kicks in a few times during the font loading. Already long ago we found out that this is also somewhat platform dependent.

	used memory	**peak memory**	**font loading time**
table	155 MB (126)	210 MB (160)	6.4 sec (6.8)
mixed	154 MB (130)	210 MB (160)	6.3 sec (6.7)
sparse	140 MB (123)	199 MB (144)	6.4 sec (6.8)

[19] For practical reasons we share as much odd as possible between the methods so some reorganization was needed.

cached	90 MB	94 MB	0.6 sec
baseline	67 MB	71 MB	0.3 sec

Here the sparse method reports less memory usage. There is no other gain as there is a lot of access to glyph data due to the fact that this font is rather advanced. More virtualization would probably work against us here.

Being a CJK font, the somewhat feature-dumb but large AdobeSongStd-Light has lots of glyphs. In previous tables we already saw values between parenthesis: these are values measured with implicit calls to the garbage collector before writing the font to the cache. For this font much more memory is used but garbage collection has a positive impact on memory consumption but drastic consequences for runtime. Eventually it's the cached timing that matters and that is a constant factor but even then it can disturb users if a first run after an update takes so much time.

	used memory	**peak memory**	**font loading time**
table	180 MB (125)	185 MB (172)	4.4 sec (4.5)
mixed	190 MB (144)	194 MB (181)	4.4 sec (4.7)
sparse	153 MB (119)	232 MB (232)	8.7 sec (8.9)
cached	96 MB	100 MB	0.7 sec
baseline	67 MB	71 MB	0.3 sec

Peak memory is quite high for the sparse method which is due to the fact that we have only glyphs (but many) so we have lots of access and small tables being created and collected. I suspect that in a regular run the loading time is much lower for the sparse case because this is just too much of a difference.

The last test loaded 40 variants of Latin Modern. Each font has reasonable number of glyphs (covering the latin script takes some 400–600 glyphs), the normal amount of kerning, but hardly any features. Reloading these 40 fonts takes about a second.

	used memory	**peak memory**	**font loading time**
table	204 MB (175)	213 MB (181)	13.1 sec (16.4)
mixed	195 MB (168)	205 MB (174)	13.4 sec (16.5)
sparse	198 MB (165)	202 MB (170)	13.4 sec (16.6)
cached	147 MB	151 MB	1.7 sec
baseline	67 MB	71 MB	0.3 sec

The old method wins in runtime and this makes it hard to decide which strategy to follow. Again the numbers between parenthesis show what happens when

we do an extra garbage collection sweep after packaging the font instance. A few more sweeps in other spots will bring down memory a few megabytes but at the cost of quite some runtime. The original module that uses the table approach is 3 seconds faster that the current one. As the code is essentially the same but organized differently again we suspect the garbage collector to be the culprit.

So when we came this far, Taco and I did some further tests and on his machine Taco ran a profiler on some of the tests. He posted the following conclusion to the LuaTEX mailing list:

> It seems that the userdata access is useful if *but only if* you are very low on memory. In other cases, it just adds extra objects to be garbage collected, which makes the collector slower. That is on top of extra time spent on the actual calls, and even worse: those extra gc objects tend to be scattered around in memory, resulting in extra minor page faults (cpu cache misses) and all that has a noticeable effect on run speed: the metatable based access is 20–30% slower than the old massive `to_table`.
>
> Therefore, there seems little point in expanding the metadata functionality any further. What is there will stay, but adding more metadata objects appears to be a waste of time on all sides.

This leaves us with a question: should we replace the old module by the experimental one? It makes sense to do this as in practice users will not be harmed much. Fonts are cached and loading a cached font is not influenced. The new module leaves the choice to the user. He or she can decide to limit memory usage (for cache building) by using directives:

```
\enabledirectives[fonts.otf.loader.method=table]
\enabledirectives[fonts.otf.loader.method=mixed]
\enabledirectives[fonts.otf.loader.method=sparse]

\enabledirectives[fonts.otf.loader.cleanup]
\enabledirectives[fonts.otf.loader.cleanup=1]
\enabledirectives[fonts.otf.loader.cleanup=2]
\enabledirectives[fonts.otf.loader.cleanup=3]
```

The cleanup has three levels and each level adds a garbage collection sweep (in a different spot). Of course three sweeps per font that is prepared for caching has quite some impact on performance. If your computer has enough memory it makes no sense to use any of these directives. For the record: these directives are not available in the generic (plain TEX) variant, at least not in the short term. As Taco mentions, cache misses can have drastic consequences and

we've ran into that years ago already when support for OpenType math was added to LuaTEX: out of a sudden and without no reason passing a font table to TEX became twice as slow on my machine. This is comparable with the new, reorganized table loader being slower than the old one. Eventually I'll get back that time, which is unlikely to happen with the unserdata variant where there is no way to bring down the number of function calls and intermediate table creation.

The previously shown values that concern all fonts including creating, caching, reloading, creating a scaled instance and passing the data to TEX. In that process quite some garbage collection can happen and that obscures the real values. However, in MkIV we report the conversion time when a font gets cached so that the user at least sees something happening. These timings are on a per font base. Watch the following values:

	table	**sparse**
song	3.2	3.6
cambria	4.9 (0.9 1.0 0.9 1.1 0.5 0.5)	5.6 (1.1 1.1 1.0 1.2 0.6 0.6)
husayni	1.2	1.3

In the case of Cambria several fonts are loaded including subfonts from TrueType containers. This shows that the table variant is definitely faster. It might be that later this is compensated by additional garbage collection but that would even worsen the sparse case were more extensive userdata be used. These values more reflect what Taco measured in the profiler. Improvements to the garbage collector are more likely to happen than a drastic speed up in function calls so the table variant is still a safe bet.

There are a few places where the renewed code can be optimized so these numbers are not definitive. Also, the loader code was not the only code adapted. As we cannot manipulate the main table in the userdata variant, the code related to patches and extra features like `tlig`, `trep` and `anum` had to be rewritten as well: more code and a bit more close to the final table format.

	table	**sparse**
hybrid	310 MB / 10.3 sec	285 MB / 10.5 sec
mk	884 MB / 47.5 sec	878 MB / 48.7 sec

The timings in the previous table concern runs of a few documents where the `mk` loads quite some large and complex fonts. The runs are times with an empty cache so all fonts are preprocessed. The memory consumption is the peak load as reported by the task manager and we need to keep in mind that Lua allocates more than it needs. Keep in mind that these values are so high because fonts

are created. A regular run takes less memory. Interesting is that for `mk` the original implementation performs better but the difference is about a second which again indicates that the garbage collector is a major factor. Timing only the total runtime gives:

	cached	**original**	**table**	**sparse**
mk	38.1 sec	75.5 sec	77.2 sec	80.8 sec

Here we used the system timer while in previous tables we used the values as reported by the timers built in MkIV (and only reported the font loading times).

The timings above are taken on my laptop running Windows 7 and this is not that good a platform for precise timings. Tacos measurements were done with specialized tools and should be trusted more. It looks indeed that the current level of userdata support is about the best compromise one can get.

In the process I also experimented with virtualizing the final TFM table, thereby simulating the upcoming virtualization of that table in LuaTEX. Interesting is that for (for instance) `mk.pdf` memory consumption went down with 20% but that document is non-typical and loades many fonts, including vitual punk fonts. However, as access to that tables happens infrequently virtualization makes muich sense there, again only at the toplevel of the characters subtable.

14.3 Hyperlinks

At Pragma ADE we have a long tradition of creating highly interactive documents. I still remember the days that processing a 20.000 page document with numerous menus and buttons on each page took a while to get finished, especially if each page has a MetaPost graphic as well.

On a regular computer a document with so many links is no real problem. After all, the PDF format is designed in such a way that only the partial content has to be loaded. However, half a million hyperlinks do demand some memory.

Recently I had to make a document that targets at one of these tablets and it is no secret that tablets (and e-readers) don't have that much memory. As in ConTEXt MkIV we have a bit more control over the backend, it will be no surprise that we are able to deal with such issues more comfortable than in MkII.

That specific document (part of a series) contained 1100 pages and each page has a navigation menu as well as an alphabetic index into the register. There

is a table of contents refering to about 200 chapters and these are backlinked to the table of contents. There are some also 200 images and tables that end up elsewhere and again are crosslinked. Of course there is the usual bunch of inline hyperlinks. So, in total this document has some 32.000 hyperlinks. The input is a 3.03 MB XML file.

	size	**one run**
don't optimize	5.76 MB	59.4 sec
prefer page references over named ones	5.66 MB	56.2 sec
agressively share similar references	5.19 MB	60.2 sec
optimize page as well as similar references	5.11 MB	56.5 sec
disable all interactive features	4.19 MB	42.7 sec

So, by aggressively sharing hyperlinks and turning all internal named destinations into page destinations we bring down the size noticeably and even have a faster run. It is for this reason that aggressive sharing is enabled by default. I you don't want it, you can disable it with:

```
\disabledirectives[refences.sharelinks]
```

Currently we use names for internal (automatically generated) links. We can force page links for them but still use names for explicit references so that we can reach them from external documents; this is called mixed mode. When no references from outside are needed, you can force pagelinks. At some point mixed mode can become the default.

```
\enabledirectives[references.linkmethod=page]
```

With values: `page`, `mixed`, `names` and `yes` being equivalent to `page`. The MKII way of setting this is still supported:

```
\setupinteraction[page=yes]
```

We could probably gain quite some more bytes by turning all repetitive elements into shared graphical objects but it only makes sense to spend time on that when a project really needs it (and pays for it). There is upto one megabyte of (compressed) data related to menus and other screen real estate that qualifies for this but it might not be worth the trouble.

The reason for trying to minimize the amount of hyperlink related metadata (in PDF terminology annotations) is that on tablets with not that much memory (and no virtual memory) we don't want to keep too much of that (redundant) data in memory. And indeed, the optimized document feels more responsive

than the dirty version, but that could as well be related to the viewing applications.

14.4 Constants

Not every optimization saves memory of runtime. They are more optimizations due to changes in circumstances. When TEX had only 256 registers one had to find ways to get round this. For instance counters are quite handy and you could quickly run out of them. In CONTEXT there are two ways to deal with this. Instead of a real count register you can use a macro:

```
\newcounter \somecounter
\increment  \somecounter
\decrement (\somecounter,4)
```

In MKIV many such pseudo counters have been replaced by real ones which is somewhat faster in usage.

Often one needs a constant and a convenient way to define such a frozen counter is:

```
\chardef \myconstant 10
\ifnum \myvariable = \myconstant ....
\ifcase \myconstant ...
```

This is both efficient and fast and works out well because TEX treats them as numbers in comparisons. However, it is somewhat clumsy, as constants have nothing to do with characters. This is why all such definitions have been replaced by:

```
\newconstant \myconstant 10
\setconstant \myconstant 12
\ifnum \myvariable = \myconstant ....
\ifcase \myconstant ...
```

We use count registers which means that when you set a constant, you can just assign the new value directly or use the `\setcounter` macro.

We already had an alternative for conditionals:

```
\newconditional \mycondition
\settrue \mycondition
\setfalse \mycondition
```

```
\ifconditional \mycondition
```

These will also be adapted to counts but first we need a new primitive.

The advantage of these changes is that at the Lua end we can consult as well as change these values. This means that in the end much more code will be adapted. Especially changing the constants resulted in quite some cosmetic changes in the core code.

14.5 Definitions

Another recent optimization was possible when at the Luaend settings lccodes cum suis and some math definitions became possible. As all these initializations take place at the Lua end till then we were just writing TEX code back to TEX, but now we stay at the Luaend. This not only looks nicer, but also results in a slightly less memory usage during format generation (a few percent). Making a format also takes a few tenths of a second less (again a few percent). The reason why less memory is needed is that instead of writing tens of thousands `\lccode` related commands to TEX we now set the value directly. As writes to TEX are collected, quite an amount of tokens get cached.

All such small improvements makes that ConTEXt MkIV runs smoother with each advance of LuaTEX. We do have a wishlist for further improvements but so far we managed to improve stepwise instead of putting too much pressure on LuaTEX development.

15 Characters with special meanings

15.1 Introduction

When TEX was designed UNICODE was not yet available and characters were encoded in a seven or eight bit encoding, like ASCII or EBCDIC. Also, the layout of keyboards was dependent of the vendor. A lot has happened since then: more and more UNICODE has become the standard (with UTF as widely used way of efficiently coding it).

Also at that time, fonts on computers were limited to 256 characters at most. This resulted in TEX macro packages dealing with some form of input encoding on the one hand and a font encoding on the other. As a side effect of character nodes storing a reference to a glyph in a font hyphenation was related to font encodings. All this was quite okay for documents written in English but when TEX became pupular in more countries more input as well as font encodings were used.

Of course, with LUATEX being a UNICODE engine this has changed, and even more because wide fonts (either TYPE1 or OPENTYPE) are supported. However, as TEX is already widely used, we cannot simply change the way characters are treated, certainly not special ones. Let's go back in time and see how plain TEX set some standards, see how CONTEXT does it currently, and look ahead how future versions will deal with it.

15.2 Catcodes

Traditional TEX is an eight bit engine while LUATEX extends this to UTF input and internally works with large numbers.

In addition to its natural number (at most 0xFF for traditional TEX and upto 0x10FFFF for LUATEX), each character can have a so called category code, or catcode. This code determines how TEX will treat the character when it is seen in the input. The category code is stored with the character so when we change such a code, already read characters retain theirs. Once typeset a character can have turned into a glyph and its catcode properties are lost.

There are 16 possible catcodes that have the following meaning:

0 escape This starts an control sequence. The scanner reads the whole sequence and stores a reference to it in an efficient way. For instance the character sequence `\relax` starts

		with a backslash that has category code zero and TEX reads on till it meets non letters. In macro definitions a reference to the so called hash table is stored.
1	begin group	This marks the begin of a group. A group an be used to indicate a scope, the content of a token list, box or macro body, etc.
2	end group	This marks the end of a group.
3	math shift	Math starts and ends with characters tagged like this. Two in a row indicate display math.
4	alignment tab	Characters with this property indicate a next entry in an alignment.
5	end line	This one is somewhat special. As line endings are operating system dependent, they are normalized to character 13 and by default that one has this category code.
6	parameter	Macro parameters start with a character with this category code. Such characters are also used in alignment specifications. In nested definitions, multiple of them in a row are used.
7	superscript	Tagged like this, a character signals that the next token (or group) is to be superscripted. Two such characters in a row will make the parser treat the following character or lowercase hexadecimal number as specification for a replacement character.
8	subscript	Codes as such, a character signals that the next token (or group) is to be subscripted.
9	ignored	When a character has this category code it is simply ignored.
10	space	This one is also special. Any character tagged as such is converted to the ASCII space character with code 32.
11	letter	Normally this are the characters that make op sequences with a meaning like words. Letters are special in the sense that macro names can only be made of letters. The hyphenation machinery will normally only deal with letters.
12	other	Examples of other characters are punctuation and special symbols.
13	active	This makes a character into a macro. Of course it needs to get a meaning in order not to trigger an error.
14	comment	All characters on the same line after comment characters are ignored.
15	invalid	An error message is issued when an invalid character is seen. This catcode is probably not assigned very often.

So, there is a lot to tell about these codes. We will not discuss the input parser here, but it is good to know that the following happens.

- The engine reads lines, and normalizes cariage return and linefeed sequences.
- Each line gets a character with number `\endlinechar` appended. Normally this is a character with code 13. In LuaTEX a value of −1 will disable this automatism.
- Normally spaces (characters with the space property) at the end of a line are discarded.
- Sequences like `^^A` are converted to characters with numbers depending on the position in ASCII vector: `^^@` is zero, `^^A` is one, etc.
- Sequences like `^^1f` are converted to characters with a number similar to the (lowercase) hexadecimal part.

Hopefully this is enough background information to get through the following sections so let's stick to a simple example:

```
\def\test#1{$x_{#1}$}
```

Here there are two control sequences, starting with a backslash with category code zero. Then comes an category 6 character that indicates a parameter that is referenced later on. The outer curly braces encapsulate the definition and the inner two braces mark the argument to a subscript, which itself is indicated by an underscore with category code 8. The start and end of mathmode is indicated with a dollar sign that is tagged as math shift (category code 3). The character `x` is just a letter.

Given the above description, how do we deal with catcodes and newlines at the LUA end? Catcodes are easy: we can print back to TEX using a specific catcode regime (later we will see a few of those regimes). As character 13 is used as default at the TEX end, we should also use it at the LUA end, i.e. we should use `\r` as line terminator (`\endlinechar`). On the other hand, we have to use `\n` (character 10, `\newlinechar`) for printing to the terminal, log file, of TEX output handles, although in CONTEXT all that happens via LUA anyway, so we don't bother too much about it here.

There is a pitfall. As TEX reads lines, it depends on the file system to provide them: it fetches lines or whatever represents the same on block devices. In LuaTEX the implementation is similar: if you plug in a reader callback, it has to provide a function that returns a line. Passing two lines does not work out as expected as TEX discards anything following the line separator (cr, lf or crlf) and then appends a normalized endline character (in our case character 13). At least, this is what TEX does naturally. So, in callbacks you can best feed line by line without any of those characters.

When you print something from LUA to TEX the situation is slightly different:

```
\startluacode
tex.print("line 1\r line 2")
tex.print("line 3\n line 4")
\stopluacode
```

This is what we get:

line 1 line 3 line 4

The explicit `\endlinechar` (`\r`) terminates the line and the rest gets discarded. However, a `\n` by default has category code 12 (other) and is turned into a space and successive spaces are (normally) ignored, which is why we get the third and fourth line separated by a space.

Things get real hairy when we do the following:

```
\startluacode
tex.print("\\bgroup")
tex.print("\\obeylines")
tex.print("line 1\r line 2")
tex.print("line 3\n line 4")
tex.print("\\egroup")
\stopluacode
```

Now we get this (the `tex.print` function appends an endline character itself):

line 1
line 2
line 3 line 4

By making the endline character active and equivalent to `\par` TEX nicely scans on and we get the second line as well. Now, if you're still with us, you're ready for the next section.

15.3 Plain TEX

In the TEX engine, some characters already have a special meaning. This is needed because otherwise we cannot use the macro language to set up the format. This is hard-coded so the next code is not really used.

```
\catcode `\^^@ =  9  % ascii null is ignored
\catcode `\^^M =  5  % ascii return is end-line
```

```
\catcode `\\   =  0  % backslash is TeX escape character
\catcode `\%   = 14  % percent sign is comment character
\catcode `\    = 10  % ascii space is blank space
\catcode `\^^? = 15  % ascii delete is invalid
```

There is no real reason for setting up the null and delete character but maybe in those days the input could contain them. The regular upper- and lowercase characters are initialized to be letters with catcode 11. All other characters get category code 12 (other).

The plain TEX format starts with setting up some characters that get a special meaning.

```
\catcode `\{   =  1 % left brace is begin-group character
\catcode `\}   =  2 % right brace is end-group character
\catcode `\$   =  3 % dollar sign is math shift
\catcode `\&   =  4 % ampersand is alignment tab
\catcode `\#   =  6 % hash mark is macro parameter character
\catcode `\^   =  7 \catcode`\^^K=7 % circumflex and uparrow
                                    % are for superscripts
\catcode `\_   =  8 \catcode`\^^A=8 % underline and downarrow
                                    % are for subscripts
\catcode `\^^I = 10 % ascii tab is a blank space
\catcode `\~   = 13 % tilde is active
```

The fact that this happens in the format file indicates that it is not by design that for instance curly braces are used for grouping, or the hash for indicating arguments. Even math could have been set up differently. Nevertheless, all macro packages have adopted these conventions so they could as well have been hard-coded presets.

Keep in mind that nothing prevents us to define more characters this way, so we could make square brackets into group characters as well. I wonder how many people have used the two additional special characters that can be used for super- and subscripts. The comment indicates that it is meant for a special keyboard.

One way to make sure that a macro will not be overloaded is to use characters in it's name that are letters when defining the macro but make sure that they are others when the user inputs text.

```
\catcode `@ = 11
```

Again, the fact that plain TEX uses the commercial at sign has set a standard. After all, at that time this symbol was not as popular as it is nowadays.

Further on in the format some more catcode magic happens. For instance this:

```
\catcode `\^^L = 13 \outer\def^^L{\par} % ascii form-feed is "\outer\par"
```

So, in your input a formfeed is equivalent to an empty line which makes sense, although later we will see that in ConTEXt we do it differently. As the tilde was already active it also gets defined:

```
\def~{\penalty10000\ } % tie
```

Again, this convention is adopted and therefore a sort of standard. Nowadays we have special Unicode characters for this, but as they don't have a visualization editing is somewhat cumbersome.

The change in catcode of the newline character `^^M` is done locally, for instance in `\obeylines`. Keep in mind that this is the character that TEX appends to the end of an input line. The space is made active when spaces are to be obeyed.

A few very special cases are the following.

```
\mathcode `\^^Z = "8000 % \ne
\mathcode `\    = "8000 % \space
\mathcode `\'   = "8000 % ^\prime
\mathcode `\_   = "8000 % \_
```

This flags those characters as being special in mathmode. Normally when you do something like this:

```
\def\test#1{$#1$} \test{x_2} \test{x''}
```

The catcodes that are set when passing the argument to `\test` are frozen when they end up in the body of the macro. This means that when `'` is other it will be other when the math list is built. However, in math mode, plain TEX wants to turn that character into a prime and even in a double one when there are two in a row. The special value `"8000` tells the math machinery that when it has an active meaning, that one will be triggered. And indeed, the plain format defined these active characters, but in a special way, sort of:

```
{ \catcode`\' = 13 \gdef'{....} }
```

So, when active it has a meaning, and it happens to be only treated as active

when in math mode.

Quite some other math codes are set as well, like:

```
\mathcode`\^^@ = "2201 % \cdot
\mathcode`\^^A = "3223 % \downarrow
\mathcode`\^^B = "010B % \alpha
\mathcode`\^^C = "010C % \beta
```

In Appendix C of The TEXbook Don Knuth explains the rationale behind this choice: he had a keyboard that has these shortcuts. As a consequence, one of the math font encodings also has that layout. It must have been a pretty classified keyboard as I could not find a picture on the internet. One can probably assemble such a keyboard from one of those keyboard that come with no imprint. Anyhow, Don explicitly says "Of course, designers of TEX macro packages that are intended to be widely used should stick to the standard ASCII characters." so that is what we do in the next sections.

15.4 How about CONTEXT

In CONTEXT we've always used several catcode regimes and switching between them was a massive operation. Think of a different regime when defining macros, inputting text, typesetting verbatim, processing XML, etc. When LUATEX introduced catcode tables, the existing mechanisms were rewritten to take advantage of this. This is the standard table for input as of December 2010.

```
\startcatcodetable \ctxcatcodes
  \catcode \tabasciicode        \spacecatcode
  \catcode \endoflineasciicode  \endoflinecatcode
  \catcode \formfeedasciicode   \endoflinecatcode
  \catcode \spaceasciicode      \spacecatcode
  \catcode \endoffileasciicode  \ignorecatcode
  \catcode \circumflexasciicode \superscriptcatcode
  \catcode \underscoreasciicode \subscriptcatcode
  \catcode \ampersandasciicode  \alignmentcatcode
  \catcode \backslashasciicode  \escapecatcode
  \catcode \leftbraceasciicode  \begingroupcatcode
  \catcode \rightbraceasciicode \endgroupcatcode
  \catcode \dollarasciicode     \mathshiftcatcode
  \catcode \hashasciicode       \parametercatcode
  \catcode \commentasciicode    \commentcatcode
  \catcode \tildeasciicode      \activecatcode
  \catcode \barasciicode        \activecatcode
```

```
\stopcatcodetable
```

Because the meaning of active characters can differ per table there is a related mechanism for switching those meanings. A careful reader might notice that the formfeed character is just a newline. If present at all, it often sits on its own line, so effectively it then behaves as in plain TEX: triggering a new paragraph. Otherwise it becomes just a space in the running text.

In addition to the active tilde we also have an active bar. This is actually one of the oldest features: we use bars for signaling special breakpoints, something that is really needed in Dutch (education), where we have many compound words. Just to show a few applications:

```
firstpart||secondpart  this|(|orthat)  one|+|two|+|three
```

In MKIV we have another way of dealing with this. There you can enable a special parser that deals with it at another level, the node list.

```
\setbreakpoints[compound]
```

When TEXies discuss catcodes some can get quite upset, probably because they spend some time fighting their side effects. Personally I like the concept. They can be a pain to deal with but also can be fun. For instance, support of XML in CONTEXT MKII was made possible by using active `<` and `&`.

When dealing with all kind of inputs the fact that characters have special meanings can get in the way. One can argue that once a few have a special meaning, it does not matter that some others have. Most complaints from users concern `$`, `&` and `_`. When for symmetry we add `^` it is clear that these characters relate to math.

Getting away from the `$` can only happen when users are willing to use for instance `\m{x}` instead of `$x$`. The `&` is an easy one because in CONTEXT we have always discouraged its use in tables and math alignments. Using (short) commands is a bit more keying but also provides more control. That leaves the `_` and `^` and there is a nice solution for this: the special math tagging discussed in the previous section.

For quite a while CONTEXT provides two commands that makes it possible to use `&`, `_` and `^` as characters with only a special meaning inside math mode. The command

```
\nonknuthmode
```

turns on this feature. The counterpart of this command is

```
\donknuthmode
```

One step further goes the command:

```
\asciimode
```

This only leave the backslash and curly braces a special meaning.

```
\startcatcodetable \txtcatcodes
  \catcode \tabasciicode       \spacecatcode
  \catcode \endoflineasciicode \endoflinecatcode
  \catcode \formfeedasciicode  \endoflinecatcode
  \catcode \spaceasciicode     \spacecatcode
  \catcode \endoffileasciicode \ignorecatcode
  \catcode \backslashasciicode \escapecatcode
  \catcode \leftbraceasciicode \begingroupcatcode
  \catcode \rightbraceasciicode\endgroupcatcode
\stopcatcodetable
```

So, even the percentage character being a comment starter is no longer there. At this time it's still being discussed where we draw the line. For instance, using the following setup renders puts TEX out of action, and we happily use it deep down in ConTEXt to deal with verbatim.

```
\startcatcodetable \vrbcatcodes
  \catcode \tabasciicode       \othercatcode
  \catcode \endoflineasciicode \othercatcode
  \catcode \formfeedasciicode  \othercatcode
  \catcode \spaceasciicode     \othercatcode
  \catcode \endoffileasciicode \othercatcode
\stopcatcodetable
```

15.5 Where are we heading?

When defining macros, in ConTEXt we not only use the `@` to provide some protection against overloading, but also the `?` and `!`. There is of course some freedom in how to use them but there are a few rules, like:

```
\c!width         % interface neutral key
\v!yes           % interface neutral value
```

```
\s!default       % system constant
\e!start         % interface specific command name snippet
\!!depth         % width as keyword to primitive
\!!stringa       % scratch macro
\??ab            % namespace
\@@abwidth       % namespace-key combination
```

There are some more but this demonstrates the principle. When defining macros that use these, you need to push and pop the current catcode regime

```
\pushcatcodes
\catcodetable \prtcatcodes
....
\popcatcodes
```

or more convenient:

```
\unprotect
....
\protect
```

Recently we introduced named parameters in ConTeXt and files that are coded that way are tagged as MkVI. Because we nowadays are less concerned about performance, some of the commands that define the user interface have been rewritten. At the cost of a bit more runtime we move towards a somewhat cleaner inheritance model that uses less memory. As a side effect module writers can define the interface to functionality with a few commands; think of defining instances with inheritance, setting up instances, accessing parameters etc. It sounds more impressive than it is in practice but the reason for mentioning it here is that this opportunity is also used to provide module writers an additional protected character: _.

```
\def\do_this_or_that#variable#index%
  {$#variable_{#index}$}

\def\thisorthat#variable#index%
  {(\do_this_or_that{#variable}{#index})}
```

Of course in the user macros we don't use the _ if only because we want that character to show up as it is meant.

```
\startcatcodetable \prtcatcodes
  \catcode \tabasciicode       \spacecatcode
```

```
  \catcode \endoflineasciicode  \endoflinecatcode
  \catcode \formfeedasciicode   \endoflinecatcode
  \catcode \spaceasciicode      \spacecatcode
  \catcode \endoffileasciicode  \ignorecatcode
  \catcode \circumflexasciicode \superscriptcatcode
  \catcode \underscoreasciicode \lettercatcode
  \catcode \ampersandasciicode  \alignmentcatcode
  \catcode \backslashasciicode  \escapecatcode
  \catcode \leftbraceasciicode  \begingroupcatcode
  \catcode \rightbraceasciicode \endgroupcatcode
  \catcode \dollarasciicode     \mathshiftcatcode
  \catcode \hashasciicode       \parametercatcode
  \catcode \commentasciicode    \commentcatcode
  \catcode `\@                  \lettercatcode
  \catcode `\!                  \lettercatcode
  \catcode `\?                  \lettercatcode
  \catcode \tildeasciicode      \activecatcode
  \catcode \barasciicode        \activecatcode
\stopcatcodetable
```

This table is currently used when defining core macros and modules. A rather special case is the circumflex. It still has a superscript related catcode, and this is only because the circumflex has an additional special meaning

Instead of the symbolic names in the previous blob of code we could have indicated characters numbers as follows:

```
\catcode `\^^I \spacecatcode
```

However, if at some point we decide to treat the circumflex similar as the underscore, i.e. give it a letter catcode, then we should not use this double circumflex method. In fact, the code base does not do that any longer, so we can decide on that any moment. If for some reason the double circumflex method is needed, for instance when defining macros like `\obeylines`, one can do this:

```
\bgroup
  \permitcircumflexescape
  \catcode \endoflineasciicode \activecatcode
  \gdef\obeylines%
    {\catcode\endoflineasciicode\activecatcode%
     \def^^M{\par}}
\egroup
```

However, in the case of a newline one can also do this:

```
\bgroup
  \catcode \endoflineasciicode \activecatcode
  \gdef\obeylines%
    {\catcode\endoflineasciicode\activecatcode%
     \def
       {\par}}
\egroup
```

Or just:

```
\def\obeylines{\defineactivecharacter 13 {\par}}
```

In ConTeXt we have the following variant, which is faster than the previous one.

```
\def\obeylines
  {\catcode\endoflineasciicode\activecatcode
   \expandafter\def\activeendoflinecode{\obeyedline}}
```

So there are not circumflexes used at all. Also, we only need to change the meaning of `\obeyedline` to give this macro another effect.

All this means that we are upgrading catcode tables, we also consider making `\nonknuthmode` the default, i.e. move the initialization to the catcode vectors. Interesting is that we could have done that long ago, as the mentioned `"8000` trickery has proven to be quite robust. In fact, in math mode we're still pretty much in knuth mode anyway.

There is one pitfall. Take this:

```
\def\test{$\something_2$} % \something_
\def\test{$\something_x$} % \something_x
```

When we are in unprotected mode, the underscore is part of the macro name, and will not trigger a subscript. The solution is simple:

```
\def\test{$\something _2$}
\def\test{$\something _x$}
```

In the rather large ConTeXt code base there were only a few spots where we had to add a space. When moving on to MkIV we have the freedom to introduce such

changes, although we don't want to break compatibility too much and only for the good. We expect this all to settle down in 2011. No matter what we decide upon, some characters will always have a special meaning. So in fact we always stay in some sort of donknuthmode, which is what TEX is all about.

16 Weird examples

16.1 Introduction

In this chapter I will collect a couple of weird examples.

16.2 Inter-character spacing

There was a discussion on the LuaTEX (dev) list about inter character spacing and ligatures. The discussion involved a mechanism inherited from pdfTEX but in ConTEXt we don't use that at all. Actually, support for inter character spacing was added in an early stage of MkIV development as an alternative for the MkII variant, which used parsing at the TEX end. Personally I never use this spacing, unless a design in a project demands it.

In the MkIV method we split ligatures when its components are known. This works quite well. It's anyway a good idea to disable ligatures, so it's more a fallback. Actually we should create components for hard coded characters like æ but as no one ever complained I leave that for a later moment.

As we already had the mechanisms in place, support for selective spacing of ligatures was a rather trivial extension. If there is ever a real need for it, I will provide control via the normal user interface, but for now using a few hooks will do. The following code shows an example of an implementation.[20]

```
local utfbyte = utf.byte
local getchar = nodes.nuts.getchar

local keep = {
    [0x0132] = true, [0x0133] = true, -- IJ ij
    [0x00C6] = true, [0x00E6] = true, -- AE ae
    [0x0152] = true, [0x0153] = true, -- OE oe
}

function typesetters.kerns.keepligature(n)
    return keep[getchar(n)]
end

local together = {
```

[20] The examples have been adapted to the latest ConTEXt where we use `\getchar (n)` instead of `n.char`.

```
    [utfbyte("c")] = { [utfbyte("k")] = true },
    [utfbyte("i")] = { [utfbyte("j")] = true },
    [utfbyte("I")] = { [utfbyte("J")] = true },
}

function typesetters.kerns.keeptogether(n1,n2)
    local k = together[getchar(n1)]
    return k and k[getchar(n2)]
end
```

The following also works:

```
local lpegmatch = lpeg.match
local fontdata  = fonts.identifiers
local getchar   = nodes.nuts.getchar
local getfont   = nodes.nuts.getfont

local keep = -- start of name
        lpeg.P("i_j")
      + lpeg.P("I_J")
      + lpeg.P("aeligature")
      + lpeg.P("AEligature")
      + lpeg.P("oeligature")
      + lpeg.P("OEligature")

function typesetters.kerns.keepligature(n)
    local d = fontdata[getfont(n)].descriptions
    local c = d and d[getchar(n)]
    local n = c and c.name
    return n and lpegmatch(keep,n)
end
```

A more generic solution would be to use the `tounicode` information, but it would be overkill as we're dealing with a rather predictable set of characters that have gotten UNICODE slots assigned. When using basemode most fonts will work anyway.

So, is this really worth the effort? Take a look at the following example.

```
\definecharacterkerning [KernMe] [factor=0.25]

\start
  \setcharacterkerning[KernMe]
```

```
  \definedfont[Serif*default]
  Ach kijk effe, \ae sop draagt een knickerbocker! \par
  \definedfont[Serif*smallcaps]
  Ach kijk effe, \ae sop draagt een knickerbocker! \par
\stop
```

Typeset this (Dutch text) looks like:

Ach kijk effe, æsop draagt een knickerbocker!

Ach kijk effe, æsop draagt een knickerbocker!

You might wonder why I decided to look into it. Right at the moment when it was discussed, I was implementing a style that needed the Calibri font that comes with MS Windows, and I visited the FontShop website to have a look at the font. To my surprise it had quite some ligatures, way more than one would expect.

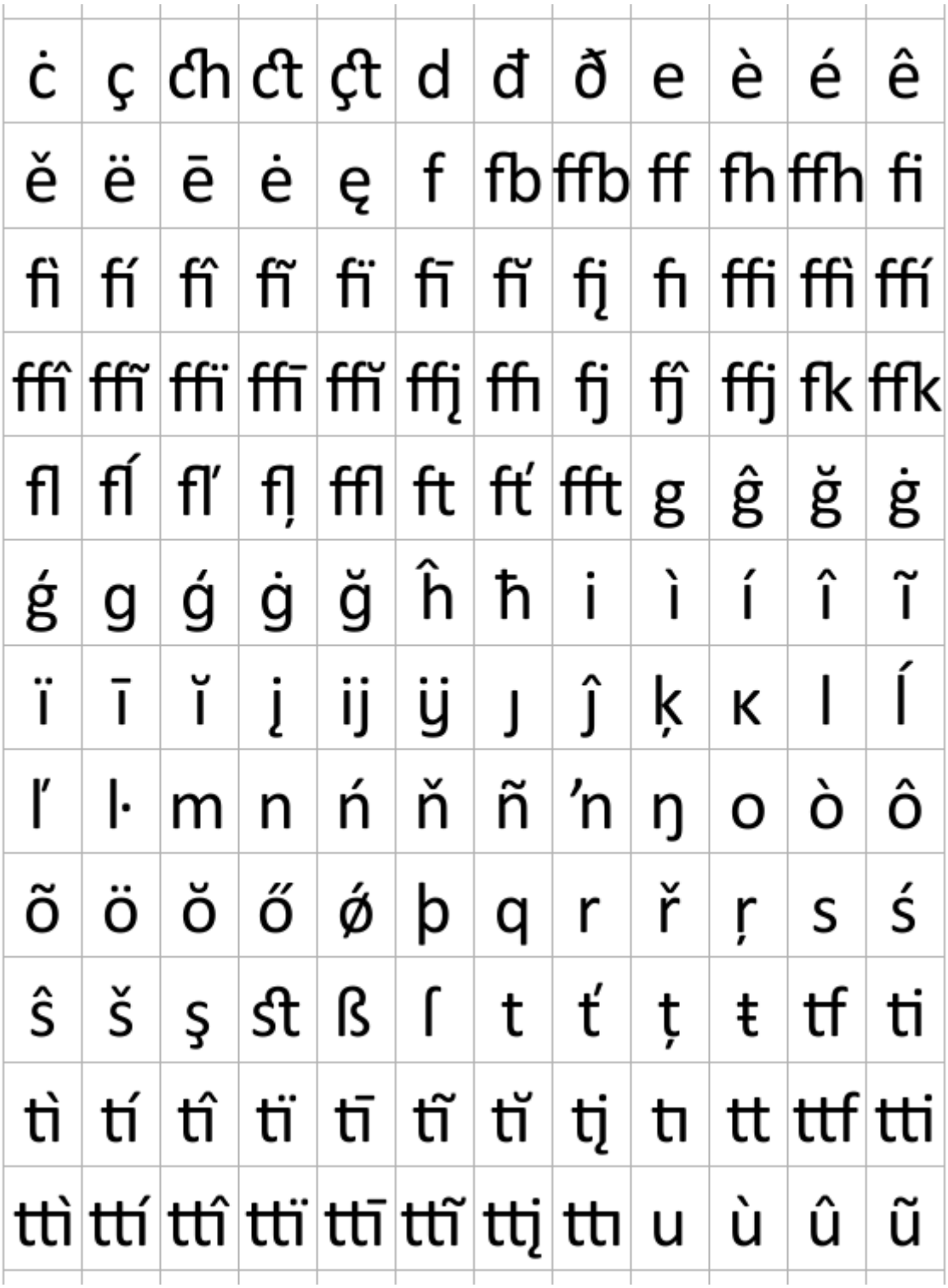


Figure 16.1 Some of the ligatures in Calibri Regular. Just wonder what intercharacter spacing will do here.

17 Glocal assignments

Here is a nice puzzle. Say that you do this:

```
\def\test{local} \test
```

What will get typeset? Right, you'll get `local`. Now take this:

```
\bgroup
  \def \test {local}[\test]
  \xdef\test{global}[\test]
  \def \test {local}[\test]
\egroup
                    [\test]
```

Will you get:

```
[local] [local] [local] [global]
```

or will it be:

```
[local] [global] [local] [global]
```

Without knowing TEX, there are good reasons for getting either of them: is a global assignment global only i.e. does it reach over the group(s) or is it global and local at the same time? The answer is that the global definitions also happens to be a local one, so the second line is what we get.

Something similar happens with registers, like counters:

```
\newcount\democount
\bgroup
         \democount 1[\the\democount]
  \global\democount 2[\the\democount]
         \democount 1[\the\democount]
\egroup
                     [\the\democount]
```

We get: `[1] [2] [1] [2]`, so this is consistent with macros. But how about boxes?

```
\bgroup
         \setbox0\hbox {local}[\copy0:\the\wd0]
```

```
  \global\setbox0\hbox{global}[\copy0:\the\wd0]
         \setbox0\hbox {local}[\copy0:\the\wd0]
\egroup
                               [\copy0:\the\wd0]
```

This gives:

```
[local:32.51053pt]
[global:39.01263pt]
[local:32.51053pt]
[global:39.01263pt]
```

Again, this is consistent, so let's do some manipulation:

```
\bgroup
         \setbox0\hbox{local}         \wd0=6em [\copy0:\the\wd0]
  \global\setbox0\hbox{global} \global\wd0=5em [\copy0:\the\wd0]
         \setbox0\hbox{local}         \wd0=6em [\copy0:\the\wd0]
\egroup
                                                [\copy0:\the\wd0]
```

```
[local     :64.79956pt]
[global   :53.99963pt]
[local     :64.79956pt]
[global   :53.99963pt]
```

Right, no surprise here, but ...

```
\bgroup
         \setbox0\hbox{local}  \wd0=6em [\copy0:\the\wd0]
  \global\setbox0\hbox{global} \wd0=5em [\copy0:\the\wd0]
         \setbox0\hbox{local}  \wd0=6em [\copy0:\the\wd0]
\egroup
                                       [\copy0:\the\wd0]
```

See the difference? There is none. The second width assignment is applied to the global box.

```
[local     :64.79956pt]
[global   :53.99963pt]
[local     :64.79956pt]
[global   :53.99963pt]
```

So how about this then:

```
\bgroup
         \setbox0\hbox{local}  \wd0=6em [\copy0:\the\wd0]
  \global\setbox0\hbox{global}          [\copy0:\the\wd0]
         \setbox0\hbox{local}  \wd0=6em [\copy0:\the\wd0]
\egroup
                                       [\copy0:\the\wd0]
```

Is this what you expect?

```
[local     :64.79956pt]
[global:39.01263pt]
[local     :64.79956pt]
[global:39.01263pt]
```

So, in the case of boxes, an assignment to a box dimension is applied to the last instance of the register, and the global nature is kind of remembered. Inside a group, registers that are accessed are pushed on a stack and the assignments are applied to the one on the stack and when no local box is assigned, the one at the outer level gets the treatment. You can also say that a global box is unreachable once a local instance is used.[21]

```
\setbox0\hbox{outer} [\copy0:\the\wd0]
\bgroup
            \wd0=6em [\copy0:\the\wd0]
\egroup
                     [\copy0:\the\wd0]
```

This gives:

```
[outer:32.51053pt]
[outer     :64.79956pt]
[outer     :64.79956pt]
```

It works as expected when we use local boxes after such an assignment:

```
\setbox0\hbox{outer}           [\copy0:\the\wd0]
\bgroup
                      \wd0=6em [\copy0:\the\wd0]
  \setbox0\hbox{inner (local)} [\copy0:\the\wd0]
```

[21] The code that implements `\global \setbox` actually removes all intermediate boxes.

```
\egroup
                                [\copy0:\the\wd0]
```

This gives:

```
[outer:32.51053pt]
[outer     :64.79956pt]
[inner (local):84.52737pt]
[outer     :64.79956pt]
```

Interestingly in practice this is natural enough not to get noticed. Also, as the TEXbook explicitly mentions that one should not mix local and global usage, not many users will do that. For instance the scratch registers 0, 2, 4, 6 and 8 are often used locally while 1, 3, 5, 7 and 9 are supposedly used global. The argument for doing this is that it does not lead to unwanted stack build-up, but the last examples given here provide another good reason. Actually, global assignments happen seldom in macro packages, at least compared to local ones.

In LUATEX we can also access boxes at the LUA end. We can for instance change the width as follows:

```
\bgroup
         \setbox0\hbox{local}
         \ctxlua{tex.box[0].width = tex.sp("6em")} [\copy0:\the\wd0]
  \global\setbox0\hbox{global}
         \ctxlua{tex.box[0].width = tex.sp("5em")} [\copy0:\the\wd0]
         \setbox0\hbox{local}
         \ctxlua{tex.box[0].width = tex.sp("6em")} [\copy0:\the\wd0]
\egroup
                                                   [\copy0:\the\wd0]
```

This is consistent with the TEX end:

```
[local     :64.79956pt]
[global   :53.99963pt]
[local     :64.79956pt]
[global   :53.99963pt]
```

This is also true for:

```
\bgroup
         \setbox0\hbox{local}
```

```
        \ctxlua{tex.box[0].width = tex.sp("6em")} [\copy0:\the\wd0]
  \global\setbox0\hbox{global}                    [\copy0:\the\wd0]
        \setbox0\hbox{local}
        \ctxlua{tex.box[0].width = tex.sp("6em")} [\copy0:\the\wd0]
\egroup
                                                  [\copy0:\the\wd0]
```

Which gives:

```
[local     :64.79956pt]
[global:39.01263pt]
[local     :64.79956pt]
[global:39.01263pt]
```

The fact that a `\global` prefix is not needed for a global assignment at the TEX end means that we don't need a special function at the LUA end for assigning the width of a box. You won't miss it.

There is one catch when coding at the TEX end. Imagine this:

```
\setbox0\hbox{local} [\copy0:\the\wd0]
\bgroup
  \wd0=6em           [\copy0:\the\wd0]
\egroup
                     [\copy0:\the\wd0]
```

In sync with what we told you will get:

```
[local:32.51053pt]
[local     :64.79956pt]
[local     :64.79956pt]
```

However, this does not look that intuitive as the following:

Here the global is redundant but it looks quite okay to put it there if only to avoid confusion.[22]

[22] I finally decided to remove some of the `\global` prefixes in my older code, but I must admit that I sometimes felt reluctant when doing it, so I kept a few.

18 Handling math: A retrospective

This is TugBoat article .. reference needed.

When you start using TEX, you cannot help but notice that math plays an important role in this system. As soon as you dive into the code you will see that there is a concept of families that is closely related to math typesetting. A family is a set of three sizes: text, script and scriptscript.

$$a^{b^c} = \frac{d}{e}$$

The smaller sizes are used in superscripts and subscripts and in more complex formulas where information is put on top of each other.

It is no secret that the latest math font technology is not driven by the TEX community but by Microsoft. They have taken a good look at TEX and extended the OpenType font model with the information that is needed to do things similar to TEX and beyond. It is a firm proof of TEX's abilities that after some 30 years it is still seen as the benchmark for math typesetting. One can only speculate what Don Knuth would have come up with if today's desktop hardware and printing technology had been available in those days.

As a reference implementation of a font Microsoft provides Cambria Math. In the specification the three sizes are there too: a font can provide specifically designed script and scriptscript variants for text glyphs where that is relevant. Control is exercised with the `ssty` feature.

Another inheritance from TEX and its fonts is the fact that larger symbols can be made out of snippets and these snippets are available as glyphs in the font, so no special additional (extension) fonts are needed to get for instance really large parentheses. The information of when to move up one step in size (given that there is a larger shape available) or when and how to construct larger symbols out of snippets is there as well. Placement of accents is made easy by information in the font and there are a whole lot of parameters that control the typesetting process. Of course you still need machinery comparable to TEX's math subsystem but Microsoft Word has such capabilities.

I'm not going to discuss the nasty details of providing math support in TEX, but rather pay some attention to an (at least for me) interesting side effect of TEX's math machinery. There are excellent articles by Bogusław Jackowski and Ulrik Vieth about how TEX constructs math and of course Knuth's publications are the ultimate source of information as well.

Even if you only glance at the implementation of traditional TEX font support, the previously mentioned families are quite evident. You can have 16 of them but 4 already have a special role: the upright roman font, math italic, math symbol and math extension. These give us access to some 1000 glyphs in theory, but when TEX showed up it was mostly a 7-bit engine and input of text was often also 7-bit based, so in practice many fewer shapes are available, and subtracting the snippets that make up the large symbols brings down the number again.

Now, say that in a formula you want to have a bold character. This character is definitely not in the 4 mentioned families. Instead you enable another one, one that is linked to a bold font. And, of course there is also a family for bold italic, slanted, bold slanted, monospaced, maybe smallcaps, sans serif, etc. To complicate things even more, there are quite a few symbols that are not covered in the foursome so we need another 2 or 3 families just for those. And yes, bold math symbols will demand even more families.

$$a + \mathbf{b} + \boldsymbol{c} = \mathtt{d} + \mathtt{e} + \ell$$

Try to imagine what this means for implementing a font system. When (in for instance ConTEXt) you choose a specific body font at a certain size, you not only switch the regular text fonts, you also initialize math. When dealing with text and a font switch there, it is no big deal to delay font loading and initialization till you really need the font. But for math it is different. In order to set up the math subsystem, the families need to be known and set up and as each one can have three members you can imagine that you easily initialize some 30 to 40 fonts. And, when you use several math setups in a document, switching between them involves at least some re-initialization of those families.

When Taco Hoekwater and I were discussing LuaTEX and especially what was needed for math, it was sort of natural to extend the number of families to 256. After all, years of traditional usage had demonstrated that it was pretty hard to come up with math font support where you could freely mix a whole regular and a whole bold set of characters simply because you ran out of families. This is a side effect of math processing happening in several passes: you can change a family definition within a formula, but as TEX remembers only the family number, a later definition overloads a previous one. The previous example in a traditional TEX approach can result in:

```
a + \fam7 b + \fam8 c = \fam9 d + \fam10 e + \fam11 f
```

Here the `a` comes from the family that reflects math italic (most likely family 1) and `+` and `=` can come from whatever family is told to provide them (this is driven by their math code properties). As family numbers are stored in the

identification pass, and in the typesetting pass resolve to real fonts you can imagine that overloading a family in the middle of a definition is not an option: it's the number that gets stored and not what it is bound to. As it is unlikely that we actually use more than 16 families we could have come up with a pool approach where families are initialized on demand but that does not work too well with grouping (or at least it complicates matters).

So, when I started thinking of rewriting the math font support for ConTeXt MkIV, I still had this nicely increased upper limit in mind, if only because I was still thinking of support for the traditional TeX fonts. However, I soon realized that it made no sense at all to stick to that approach: OpenType math was on its way and in the meantime we had started the math font project. But given that this would easily take some five years to finish, an intermediate solution was needed. As we can make virtual fonts in LuaTeX, I decided to go that route and for several years already it has worked quite well. For the moment the traditional TeX math fonts (Computer Modern, px, tx, Lucida, etc) are virtualized into a pseudo-OpenType font that follows the Unicode math standard. So instead of needing more families, in ConTeXt we could do with less. In fact, we can do with only two: one for regular and one for bold, although, thinking of it, there is nothing that prevents us from mixing different font designs (or preferences) in one formula but even then a mere four families would still be fine.

To summarize this, in ConTeXt MkIV the previous example now becomes:

```
U+1D44E + U+1D41B + 0x1D484 = U+1D68D + U+1D5BE + U+1D4BB
```

For a long time I have been puzzled by the fact that one needs so many fonts for a traditional setup. It was only after implementing the ConTeXt MkIV math subsystem that I realized that all of this was only needed in order to support alphabets, i.e. just a small subset of a font. In Unicode we have quite a few math alphabets and in ConTeXt we have ways to map a regular keyed-in (say) 'a' onto a bold or monospaced one. When writing that code I hadn't even linked the Unicode math alphabets to the family approach for traditional TeX. Not being a mathematician myself I had no real concept of systematic usage of alternative alphabets (apart from the occasional different shape for an occasional physics entity).

Just to give an idea of what Unicode defines: there are alphabets in regular (upright), bold, italic, bold italic, script, bold script, fraktur, bold fraktur, double-struck, sans-serif, sans-serif bold, sans-serif italic, sans-serif bold italic and monospace. These are regular alphabets with upper- and lowercase characters complemented by digits and occasionally Greek.

It was a few years later (somewhere near the end of 2010) that I realized that a lot of the complications in (and load on) a traditional font system were simply due to the fact that in order to get one bold character, a whole font had to be loaded in order for families to express themselves. And that in order to have several fonts being rendered, one needed lots of initialization for just a few cases. Instead of wasting one font and family for an alphabet, one could as well have combined 9 (upper and lowercase) alphabets into one font and use an offset to access them (in practice we have to handle the digits too). Of course that would have meant extending the TEX math machinery with some offset or alternative to some extensive mathcode juggling but that also has some overhead.

If you look at the plain TEX definitions for the family related matters, you can learn a few things. First of all, there are the regular four families defined:

```
\textfont0=\tenrm \scriptfont0=\sevenrm \scriptscriptfont0=\fiverm
\textfont1=\teni  \scriptfont1=\seveni  \scriptscriptfont1=\fivei
\textfont2=\tensy \scriptfont2=\sevensy \scriptscriptfont2=\fivesy
\textfont3=\tenex \scriptfont3=\tenex   \scriptscriptfont3=\tenex
```

Each family has three members. There are some related definitions as well:

```
\def\rm       {\fam0\tenrm}
\def\mit      {\fam1}
\def\oldstyle{\fam1\teni}
\def\cal      {\fam2}
```

So, with `\rm` you not only switch to a family (in math mode) but you also enable a font. The same is true for `\oldstyle` and this actually brings us to another interesting side effect. The fact that oldstyle numerals come from a math font has implications for the way this rendering is supported in macro packages. As naturally all development started when TEX came around, package design decisions were driven by the basic fact that there was only one math font available. And, as a consequence most users used the Computer Modern fonts and therefore there was never a real problem in getting those oldstyle characters in your document.

However, oldstyle figures are a property of a font design (like table digits) and as such not specially related to math. And, why should one tag each number then? Of course it's good practice to tag extensively (and tagging makes switching fonts easy) but to tag each number is somewhat over the top. When more fonts (usable in TEX) became available it became more natural to use a proper oldstyle font for text and the `\oldstyle` more definitely ended up as a math command. This was not always easy to understand for users who primarily used TEX for anything but math.

Another interesting aspect is that with OpenType fonts oldstyle figures are again an optional feature, but now at a different level. There are a few more such traditional issues: bullets often come from a math font as well (which works out ok as they have nice, not so tiny bullets). But the same is true for triangles, squares, small circles and other symbols. And, to make things worse, some come from the regular TEX math fonts, and others from additional ones, like the ams symbols. Again, OpenType and Unicode will change this as now these symbols are quite likely to be found in fonts as they have a larger repertoire of shapes.

From the perspective of going from MkII to MkIV it boils down to changing old mechanisms that need to handle all this (dependent on the availability of fonts) to cleaner setups. Of course, as fonts are never completely consistent, or complete for that matter, and features can be implemented incorrectly or incompletely we still end up with issues, but (at least in ConTEXt) dealing with that has been moved to runtime manipulation of the fonts themselves (as part of the so-called font goodies).

Back to the plain definitions, we now arrive at some new families:

```
\newfam\itfam \def\it{\fam\itfam\tenit}
\newfam\slfam \def\sl{\fam\slfam\tensl}
\newfam\bffam \def\bf{\fam\bffam\tenbf}
\newfam\ttfam \def\tt{\fam\ttfam\tentt}
```

The plain TEX format was never meant as a generic solution but instead was an example of a macro set and serves as a basis for styles used by Don Knuth for his books. Nevertheless, in spite of the fact that TEX was made to be extended, pretty soon it became frozen and the macros and font definitions that came with it became the benchmark. This might be the reason why Unicode now has a monospaced alphabet. Once you've added monospaced you might as well add more alphabets as for sure in some countries they have their own preferences.[23]

As with `\rm`, the related commands are meant to be used in text as well. More interesting is to see what follows now:

```
\textfont        \itfam=\tenit
\textfont        \slfam=\tensl
```

[23] At the Dante 2011 meeting we had interesting discussions during dinner about the advantages of using Sütterlinschrift for vector algebra and the possibilities for providing it in the upcoming TEX Gyre math fonts.

```
\textfont        \bffam=\tenbf
\scriptfont      \bffam=\sevenbf
\scriptscriptfont\bffam=\fivebf

\textfont        \ttfam=\tentt
```

Only the bold definition has all members. This means that (regular) italic, slanted, and monospaced are not actually that much math at all. You will probably only see them in text inside a math formula. From this you can deduce that contrary to what I said before, these variants were not really meant for alphabets, but for text in which case we need complete fonts. So why do I still conclude that we don't need all these families? In practice text inside math is not always done this way but with a special set of text commands. This is a consequence of the fact that when we add text, we want to be able to do so in each language with even language-specific properties supported. And, although a family switch like the above might do well for English, as soon as you want Polish (extended Latin), Cyrillic or Greek you definitely need more than a family switch, if only because encodings come into play. In that respect it is interesting that we do have a family for monospaced, but that `\Im` and `\Re` have symbolic names, although a more extensive setup can have a blackboard family switch.

By the way, the fact that TEX came with italic alongside slanted also has some implications. Normally a font design has either italic or something slanted (then called oblique). But, Computer Modern came with both, which is no surprise as there is a metadesign behind it. And therefore macro packages provide ways to deal with those variants alongside. I wonder what would have happened if this had not been the case. Nowadays there is always this regular, italic (or oblique), bold and bold italic set to deal with, and the whole set can become lighter or bolder.

In ConTEXt MkII, however, the set is larger as we also have slanted and bold slanted and even smallcaps, so most definition sets have 7 definitions instead of 4. By the way, smallcaps is also special. if Computer Modern had had smallcaps for all variants, support for them in ConTEXt undoubtedly would have been kept out of the mentioned 7 but always been a new typeface definition (i.e. another fontclass for insiders). So, when something would have to be smallcaps, one would simply switch the whole lot to smallcaps (bold smallcaps, etc.). Of course this is what normally happens, at least in my setups, but nevertheless one can still find traces of this original Computer Modern-driven approach. And now we are at it: the whole font system still has the ability to use design sizes and combine different ones in sets, if only because in Computer Modern you don't have all sizes. The above definitions use ten, seven and five, but for

instance for an eleven point set up you need to creatively choose the proper originals and scale them to the right family size. Nowadays only a few fonts ship with multiple design sizes, and although some can be compensated with clever hinting it is a pity that we can apply this mechanism only to the traditional TEX fonts.

Concerning the slanting we can remark that TEXies are so fond of this that they even extended the TEX engines to support slanting in the core machinery (or more precisely in the backend while the frontend then uses adapted metrics). So, slanting is available for all fonts.

This brings me to another complication in writing a math font subsystem: bold. During the development of ConTEXt MkII I was puzzled by the fact that user demands with respect to bold were so inconsistent. This is again related to the way a somewhat simple setup looks: explicitly switching to bold characters or symbols using a `\bf` (alike) switch. This works quite well in most cases, but what if you use math in a section title? Then the whole lot should be in bold and an embedded bold symbol should be heavy (i.e. more bold than bold). As a consequence (and due to limited availability of complete bold math fonts) in MkII there are several bold strategies implemented.

However, in a Unicode universe things become surprisingly easy as Unicode defines those symbols that have bold companions (whatever you want to call them, mostly math alphanumerics) so a proper math font has them already. This limited subset is often available in a font collection and font designers can stick to that subset. So, eventually we get one regular font (with some bold glyphs according to the Unicode specification) and a bold companion that has heavy variants for those regular bold shapes.

The simple fact that Unicode distinguishes regular and bold simplifies an implementation as it's easier to take that as a starting point than users who for all their goodwill see only their small domain of boldness.

It might sound like Unicode solves all our problems but this is not entirely true. For instance, the Unicode principle that no character should be there more than once has resulted in holes in the Unicode alphabets, especially Greek, blackboard, fraktur and script. As exceptions were made for non-math I see no reason why the few math characters that now put holes in an alphabet could not have been there. As with more standards, following some principles too strictly eventually results in all applications that follow the standard having to implement the same ugly exceptions explicitly. As some standards aim for longevity I wonder how many programming hours will be wasted this way.

This brings me to the conclusion that in practice 16 families are more than

enough in a UNICODE-aware TEX engine especially when you consider that for a specific document one can define a nice set of families, just as in plain TEX. It's simply the fact that we want to make a macro package that does it all and therefore has to provide all possible math demands into one mechanism that complicates life. And the fact that UNICODE clearly demonstrates that we're only talking about alphabets has brought (at least) CONTEXT back to its basics: a relatively simple, few-family approach combined with a dedicated alphabet selection system. Of course eventually users may come up with new demands and we might again end up with a mess. After all, it's the fact that TEX gives us control that makes it so much fun.

19 Exporting math

19.1 Introduction

As ConTeXt has an xml export feature and because TeX is often strongly associated with math typesetting, it makes sense to take a look at coding and exporting math. In the next sections some aspects are discussed. The examples shown are a snaphot of the possibilities around June 2011.

19.2 Encoding the math

In ConTeXt there are several ways to input math. In the following example we will use some bogus math with enough structure to get some interesting results.

The most natural way to key in math is using the TeX syntax. Of course you need to know the right commands for accessing special symbols, but if you're familiar with a certain domain, this is not that hard.

```
\startformula
  \frac { x \geq 2 } { y \leq 4 }
\stopformula
```

$$\frac{x \geq 2}{y \leq 4}$$

When you have an editor that can show more than ascii the following also works out well.

```
\startformula
  \frac { x ≥ 2 } { y ≤ 4 }
\stopformula
```

One can go a step further and use the proper math italic alphabet but there are hardly any (monospaced) fonts out there that can visualize it.

```
\startformula
  \frac { 𝑥 ≥ 2 } { 𝑦 ≤ 4 }
\stopformula
```

Anyhow, ConTeXt is quite capable of remapping the regular alphabets onto the real math ones, so you can stick to `x` and `y`.

Another way to enter the same formula is by using what we call calculator math. We came up with this format many years ago when ConTeXt had to process student input using a syntax similar to what the calculators they use at school accept.

```
\startformula
  \calcmath{(x >= 2)/(y <= 4)}
\stopformula
```

$$\frac{x \geq 2}{y \leq 4}$$

As ConTeXt is used in a free and open school math project, and because some of our projects mix MathML into xml encoded sources, we can also consider using MathML. The conceptually nicest way is to use content markup, where the focus is on meaning and interchangability and not on rendering. However, we can render it quite well. OpenMath, now present in MathML 3 is also supported.

```
<math xmlns='http://www.w3c.org/mathml' version='2.0'>
  <apply> <divide/>
    <apply> <geq/> <ci> x </ci> <cn> 2 </cn> </apply>
    <apply> <leq/> <ci> y </ci> <cn> 4 </cn> </apply>
  </apply>
</math>
```

$$\frac{x \geq 2}{y \leq 4}$$

In practice MathML will be coded using the presentational variant. In many aspects this way of coding is not much different from what TeX does.

```
<math xmlns='http://www.w3c.org/mathml' version='2.0'>
  <mfrac>
    <mrow> <mi> x </mi> <mo> &geq; </mo> <mn> 2 </mn> </mrow>
    <mrow> <mi> y </mi> <mo> &leq; </mo> <mn> 4 </mn> </mrow>
  </mfrac>
</math>
```

$$\frac{x \geq 2}{y \leq 4}$$

When we enable xml export in the backend of ConTeXt, all of the above variants are converted into the following:

```
<m:math display="block">
  <m:mrow>
```

```
    <m:mfrac>
      <m:mrow>
        <m:mi>x</m:mi>
        <m:mo>≥</m:mo>
        <m:mn>2</m:mn>
      </m:mrow>
      <m:mrow>
        <m:mi>y</m:mi>
        <m:mo>≤</m:mo>
        <m:mn>4</m:mn>
      </m:mrow>
    </m:mfrac>
  </m:mrow>
</m:math>
```

This is pretty close to what we have entered as presentation MathML. The main difference is that the (display or inline) mode is registered as attribute and that entities have been resolved to UTF. Of course one could use UTF directly in the input.

19.3 Parsing the input

In TEX typesetting math happens in two stages. First the input is parsed and converted into a so called math list. In the following case it's a rather linear list, but in the case of a fraction it is a tree.

```
\startformula
  x = - 1.23
\stopformula
```

$$x = -1.23$$

A naive export looks as follows. The sequence becomes an `mrow`:

```
<m:math display="block">
  <m:mrow>
    <m:mi>x</m:mi>
    <m:mo>=</m:mo>
    <m:mo>-</m:mo>
    <m:mn>1</m:mn>
    <m:mo>.</m:mo>
    <m:mn>2</m:mn>
```

```
    <m:mn>3</m:mn>
  </m:mrow>
</m:math>
```

However, we can clean this up without too much danger of getting invalid output:

```
<m:math display="block">
  <m:mrow>
    <m:mi>x</m:mi>
    <m:mo>=</m:mo>
    <m:mo>-</m:mo>
    <m:mn>1.23</m:mn>
  </m:mrow>
</m:math>
```

This is still not optimal, as one can argue that the minus sign is part of the number. This can be taken care of at the input end:

```
\startformula
  x = \mn{- 1.23}
\stopformula
```

Now we get:

```
<m:math display="block">
  <m:mrow>
    <m:mi>x</m:mi>
    <m:mo>=</m:mo>
    <m:mn>-1.23</m:mn>
  </m:mrow>
</m:math>
```

Tagging a number makes sense anyway, for instance when we use different numbering schemes:

```
\startformula
  x = \mn{0x20DF} = 0x20DF
\stopformula
```

We get the first number nicely typeset in an upright font but the second one becomes a mix of numbers and identifiers:

$$x = 0\text{x}20\text{DF} = 0x20DF$$

This is nicely reflected in the export:

```
<m:math display="block">
  <m:mrow>
    <m:mi>x</m:mi>
    <m:mo>=</m:mo>
    <m:mn>0x20DF</m:mn>
    <m:mo>=</m:mo>
    <m:mn>0</m:mn>
    <m:mi>x</m:mi>
    <m:mn>20</m:mn>
    <m:mi>D</m:mi>
    <m:mi>F</m:mi>
  </m:mrow>
</m:math>
```

In a similar fashion we can use `\mo` and `\mi` although these are seldom needed, if only because characters and symbols already carry these properties with them.

19.4 Enhancing the math list

When the input is parsed into a math list the individual elements are called noads. The most basic noad has pointers to a nucleus, a superscript and a subscript and each of them can be the start of a sublist. All lists (with more than one character) are quite similar to `mrow` in MathML. In the export we do some flattening because otherwise we would get too many redundant `mrow`s, not that it hurts but it saves bytes.

```
\startformula
  x_n^2
\stopformula
```

This renders as:

$$x_n^2$$

And it gets exported as:

```
<m:math display="block">
  <m:mrow>
```

```
    <m:msubsup>
      <m:mi>x</m:mi>
      <m:mi>n</m:mi>
      <m:mn>2</m:mn>
    </m:msubsup>
  </m:mrow>
</m:math>
```

As said, in the math list this looks more or less the same: we have a noad with a nucleus pointing to a math character (x) and two additional pointers to the sub- and superscripts.

After this math list is typeset, we will end up with horizontal and vertical lists with glyphs, kerns, glue and other nodes. In fact we end up with what can be considered regular references to slots in a font mixed with positioning information. In the process the math properties gets lost. This happens between step 3 and 4 in the next overview.

1	XML	optional alternative input
2	TEX	native math coding
3	noads	intermediate linked list / tree
4	nodes	linked list with processed (typeset) math
5a	PDF	page description suitable for rendering
5b	XML	export reflecting the final document content

In CONTEXT MKIV we intercept the math list (with noads) and apply a couple of manipulations to it, most noticeably relocation of characters. Last in the (currently some 10) manipulation passes over the math list comes tagging. This only happens when the export is active or when we produce tagged pdf.[24]

By tagging the recognizable math snippets we can later use those persistent properties to reverse engineer the MATHML from the input.

19.5 Intercepting the typeset content

When a page gets shipped out, we also convert the typeset content to an intermediate form, ready for export later on. Version 0.22 of the exporter has a rather verbose tracing mechanism and the simple example with sub- and superscript is reported as follows:

[24] Currently the export is the benchmark and the tagged PDF implementation follows, so there can be temporary incompatibilities.

```
<math-8 trigger='268' index='1'>
  <mrow-20 trigger='268' index='1'>
    <msubsup-1 trigger='268' index='1'>
      <mi-15 trigger='268' index='1'>
        <!-- processing glyph 2 (tag 270) -->
        <!-- moving from depth 11 to 11 (mi-15) -->
        <!-- staying at depth 11 (mi-15) -->
        <!-- start content with length 4 -->
        x
        <!-- stop content -->
        <!-- moving from depth 11 to 11 (mn-13) -->
      </mi-15>
      <mn-13 trigger='270' index='2'>
        <!-- processing kern > threshold (tag 270 => 267)
        <!-- moving from depth 11 to 11 (mn-13) -->
        <!-- staying at depth 11 (mn-13) -->
        <!-- start content with length 1 -->
        2
        <!-- stop content -->
        <!-- injecting spacing 9 -->
        <!-- moving from depth 11 to 10 (msubsup-1) -->
      </mn-13>
    </msubsup-1>
    <!-- processing glyph  (tag 269) -->
    <!-- moving from depth 9 to 10 (msubsup-1) -->
    <msubsup-1 trigger='267' index='2'>
      <!-- start content with length 1 -->

      <!-- stop content -->
    </msubsup-1>
    <!-- moving from depth 9 to 11 (mi-16) -->
    <msubsup-1 trigger='269' index='3'>
      <mi-16 trigger='269' index='1'>
        <!-- processing glue > threshold (tag 269 => 262) -->
        <!-- moving from depth 11 to 11 (mi-16) -->
        <!-- staying at depth 11 (mi-16) -->
        <!-- start content with length 4 -->
        n
        <!-- stop content -->
        <!-- injecting spacing 6 -->
        <!-- moving from depth 11 to 6 (formula-8) -->
      </mi-16>
    </msubsup-1>
```

```
  </mrow-20>
</math-8>
```

This is not yet what we want so some more effort is needed in order to get proper MathML.

19.6 Exporting the result

The report that we showed before representing the simple example with super- and subscripts is strongly related to the visual rendering. It happens that TEX first typesets the superscript and then deals with the subscript. Some spacing is involved which shows up in the report between the two scripts.

In MathML we need to swap the order of the scripts, so effectively we need:

```
<math-8 trigger='268' index='1'>
  <mrow-20 trigger='268' index='1'>
    <msubsup-1 trigger='268' index='1'>
      <mi-15 trigger='268' index='1'>
        x
      </mi-15>
      <mi-16 trigger='269' index='2'>
        n
      </mi-16>
      <mn-13 trigger='270' index='3'>
        2
      </mn-13>
    </msubsup-1>
  </mrow-20>
</math-8>
```

This swapping (and some further cleanup) is done before the final tree is written to a file. There we get:

```
<m:math display="block">
  <m:mrow>
    <m:msubsup>
      <m:mi>x</m:mi>
      <m:mi>n</m:mi>
      <m:mn>2</m:mn>
    </m:msubsup>
  </m:mrow>
```

```
</m:math>
```

This looks pretty close to the intermediate format. In case you wonder with how much intermediate data we end up, the answer is: quite some. The reason will be clear: we intercept typeset output and reconstruct the input from that, which means that we have additional information travelling with the content. Also, we need to take crossing pages into account and we need to reconstruct paragraphs. There is also some overhead in making the XML look acceptable but that is neglectable. In terms of runtime, the overhead of an export (including tagging) is some 10% which is not that bad, and there is some room for optimization.

19.7 Special treatments

In content MathML the `apply` tag is the cornerstone of the definition. Because there is enough information the rendering mechanism can deduce when a function is applied and act accordingly when it comes to figuring out the right amount of spacing. In presentation MathML there is no such information and there the signal is given by putting a character with code `U+2061` between the function identifier and the argument. In TEX input all this is dealt with in the macro that specifies a function but some ambiguity is left.

Compare the following two formulas:

```
\startformula
  \tan = \frac { \sin } { \cos }
\stopformula
```

$$\tan = \frac{\sin}{\cos}$$

In the export this shows up as follows:

```
<m:math display="block">
  <m:mrow>
    <!-- begin function -->
      <m:mi>tan</m:mi>
    <!-- end function -->
    <m:mo>=</m:mo>
    <m:mrow>
      <m:mfrac>
        <m:mrow>
          <!-- begin function -->
            <m:mi>sin</m:mi>
```

```
          <!-- end function -->
        </m:mrow>
        <m:mrow>
          <!-- begin function -->
            <m:mi>cos</m:mi>
          <!-- end function -->
        </m:mrow>
      </m:mfrac>
    </m:mrow>
  </m:mrow>
</m:math>
```

Watch how we know that `tan` is a function and not a multiplication of the variables `t`, `a` and `n`.

In most cases functions will get an argument, as in:

```
\startformula
  \tan (x) = \frac { \sin (x) } { \cos (x) }
\stopformula
```

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

```
<m:math display="block">
  <m:mrow>
    <!-- begin function -->
      <m:mi>tan</m:mi>
    <!-- end function -->
    <m:mo>(</m:mo>
    <m:mi>x</m:mi>
    <m:mo>)</m:mo>
    <m:mo>=</m:mo>
    <m:mrow>
      <m:mfrac>
        <m:mrow>
          <!-- begin function -->
            <m:mi>sin</m:mi>
          <!-- end function -->
          <m:mo>(</m:mo>
          <m:mi>x</m:mi>
          <m:mo>)</m:mo>
        </m:mrow>
        <m:mrow>
```

```
          <!-- begin function -->
            <m:mi>cos</m:mi>
          <!-- end function -->
          <m:mo>(</m:mo>
          <m:mi>x</m:mi>
          <m:mo>)</m:mo>
        </m:mrow>
      </m:mfrac>
    </m:mrow>
  </m:mrow>
</m:math>
```

As expected we now see the arguments but it is still not clear that the function has to be applied.

```
\startformula
  \apply \tan {(x)} = \frac {
    \apply \sin {(x)}
  } {
    \apply \cos {(x)}
  }
\stopformula
```

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

This time we get the function application signal in the output. We could add it automatically in some cases but for the moment we don't do so. Because this trigger has no visual rendering and no width it will not be visible in an editor. Therefore we output an entity.

```
<m:math display="block">
  <m:mrow>
    <m:mi>tan</m:mi>
    <m:mo>&#x2061;</m:mo>
    <m:mo>(</m:mo>
    <m:mi>x</m:mi>
    <m:mo>)</m:mo>
    <m:mo>=</m:mo>
    <m:mrow>
      <m:mfrac>
        <m:mrow>
          <m:mi>sin</m:mi>
          <m:mo>&#x2061;</m:mo>
```

```
          <m:mo>(</m:mo>
          <m:mi>x</m:mi>
          <m:mo>)</m:mo>
        </m:mrow>
        <m:mrow>
          <m:mi>cos</m:mi>
          <m:mo>&#x2061;</m:mo>
          <m:mo>(</m:mo>
          <m:mi>x</m:mi>
          <m:mo>)</m:mo>
        </m:mrow>
      </m:mfrac>
    </m:mrow>
  </m:mrow>
</m:math>
```

In the future, we will extend the `\apply` macro to also deal with automatically managed fences. Talking of those, fences are actually supported when explicitly coded:

```
\startformula
  \apply \tan {\left(x\right)} = \frac {
    \apply \sin {\left(x\right)}
  } {
    \apply \cos {\left(x\right)}
  }
\stopformula
```

$$\tan\left(x\right) = \frac{\sin\left(x\right)}{\cos\left(x\right)}$$

This time we get a bit more structure because delimiters in TEX can be recognized easily. Of course it helps that in ConTEXt we already have the infrastructure in place.

```
<m:math display="block">
  <m:mrow>
    <m:mi>tan</m:mi>
    <m:mo>&#x2061;</m:mo>
    <m:mrow>
      <m:mfenced left="(" right=")">
        <m:mi>x</m:mi>
      </m:mfenced>
    </m:mrow>
```

```
    <m:mo>=</m:mo>
    <m:mrow>
      <m:mfrac>
        <m:mrow>
          <m:mi>sin</m:mi>
          <m:mo>&#x2061;</m:mo>
          <m:mfenced left="(" right=")">
            <m:mi>x</m:mi>
          </m:mfenced>
        </m:mrow>
        <m:mrow>
          <m:mi>cos</m:mi>
          <m:mo>&#x2061;</m:mo>
          <m:mfenced left="(" right=")">
            <m:mi>x</m:mi>
          </m:mfenced>
        </m:mrow>
      </m:mfrac>
    </m:mrow>
  </m:mrow>
</m:math>
```

Yet another special treatment is needed for alignments. We use the next example to show some radicals as well.

```
\startformula
  \startalign
    \NC  a^2 \EQ \sqrt{b}    \NR
    \NC  c   \EQ \frac{d}{e} \NR
    \NC      \EQ f           \NR
  \stopalign
\stopformula
```

It helps that in ConTeXt we use a bit of structure in math alignments. In fact, a math alignment is just a regular alignment, with math in its cells. As with other math, eventually we end up with boxes so we need to make sure that enough information is passed along to reconstuct the original.

$$\begin{aligned} a^2 &= \sqrt{b} \\ c &= \frac{d}{e} \\ &= f \end{aligned}$$

```
<m:math display="inline">
  <m:mtable detail='align'>
    <m:mtr>
      <m:mtd>
        <m:mrow>
          <m:msup>
            <m:mi>a</m:mi>
            <m:mn>2</m:mn>
          </m:msup>
        </m:mrow>
      </m:mtd>
      <m:mtd>
        <m:mrow>
          <m:mo>=</m:mo>
          <m:mroot>
            <m:mi>b</m:mi>
          </m:mroot>
        </m:mrow>
      </m:mtd>
    </m:mtr>
    <m:mtr>
      <m:mtd>
        <m:mrow>
          <m:mi>c</m:mi>
        </m:mrow>
      </m:mtd>
      <m:mtd>
        <m:mrow>
          <m:mo>=</m:mo>
          <m:mfrac>
            <m:mrow>
              <m:mi>d</m:mi>
            </m:mrow>
            <m:mrow>
              <m:mi>e</m:mi>
            </m:mrow>
          </m:mfrac>
        </m:mrow>
      </m:mtd>
    </m:mtr>
    <m:mtr>
      <m:mtd>
        <m:mrow>
```

```
          <m:mo>=</m:mo>
          <m:mi>f</m:mi>
        </m:mrow>
      </m:mtd>
    </m:mtr>
  </m:mtable>
</m:math>
```

Watch how the equal sign ends up in the cell. Contrary to what you might expect, the relation symbols (currently) don't end up in their own column. Keep in mind that these tables look structured but that presentational MathML does not assume that much structure.[25]

19.8 Units

Rather early in the history of ConTeXt we had support for units and the main reason for this was that we wanted consistent spacing. The input of the old method looks as follows:

```
10 \Cubic \Meter \Per \Second
```

This worked in regular text as well as in math and we even have an xml variant. A few years ago I played with a different method and the Lua code has been laying around for a while but never made it into the ConTeXt core. However, when playing with the export, I decided to pick up that thread. The verbose variant can now be coded as:

```
10 \unit{cubic meter per second}
```

but equally valid is:

```
10 \unit{m2/s}
```

and also

```
\unit{10 m2/s}
```

is okay. So, one can use the short (often official) symbols as well as more verbose names. In order to see what gets output we cook up some bogus units.

[25] The spacing could be improved here but it's just an example, not something real.

```
30 \unit{kilo pascal square meter / kelvin second}
```

This gets rendered as: 30 kPa·m^2/K·s . The export looks as follows:

```
30 <unit>kPa·m<sup>2</sup>/K·s</unit>
```

You can also say:

```
\unit{30 kilo pascal square meter / kelvin second}
```

and get: 30 kPa·m^2/K·s . This time the export looks like this:

```
<quantity>
  <number>30</number>
  <unit>kPa·m<sup>2</sup>/K·s</unit>
</quantity>
```

When we use units in math, the rendering is mostly the same. So,

```
$30 \unit{kilo pascal square meter / kelvin second }$
```

Gives: $30\mathrm{kPa{\cdot}m^2/K{\cdot}s}$, but the export now looks different:

```
<m:math display="inline">
  <m:mrow>
  <m:mn>30</m:mn>
  <m:maction actiontype="unit">
    <m:mrow>
      <m:mi mathvariant="normal">k</m:mi>
      <m:mi mathvariant="normal">P</m:mi>
      <m:mi mathvariant="normal">a</m:mi>
      <m:mo>·</m:mo>
      <m:msup>
          <m:mi mathvariant="normal">m</m:mi>
          <m:mn>2</m:mn>
      </m:msup>
      <m:mo>/</m:mo>
      <m:mi mathvariant="normal">K</m:mi>
      <m:mo>·</m:mo>
      <m:mi mathvariant="normal">s</m:mi>
    </m:mrow>
  </m:maction>
  </m:mrow>
</m:math>
```

Watch how we provide some extra information about it being a unit and how the rendering is controlled as by default a renderer could turn the `K` and other identifiers into math italic. Of course the subtle spacing is lost as we assume a clever renderer that can use the information provided in the `maction`.

19.9 Conclusion

So far the results of the export look quite acceptable. It is to be seen to what extent typographic detail will be added. Thanks to Unicode math we don't need to add style directives. Because we carry information with special spaces, we could add these details if needed but for the moment the focus is on getting the export robust on the one end, and extending ConTeXt's math support with some additional structure.

The export shows in the previous sections was not entirely honest: we didn't show the wrapper. Say that we have this:

```
\startformula
  e = mc^2
\stopformula
```

This shows up as:

$$e = mc^2$$

and exports as:

```
<formula>
  <formulacontent>
    <m:math display="block">
      <m:mrow>
        <m:mi>e</m:mi>
        <m:mo>=</m:mo>
        <m:mi>m</m:mi>
        <m:msup>
          <m:mi>c</m:mi>
          <m:mn>2</m:mn>
        </m:msup>
      </m:mrow>
    </m:math>
  </formulacontent>
</formula>
```

```
\placeformula
  \startformula
     e = mc^2
  \stopformula
```

This becomes:

$$e = mc^2 \tag{19.1}$$

and exports as:

```
<formula>
  <formulacontent>
    <m:math display="block">
      <m:mrow>
        <m:mi>e</m:mi>
        <m:mo>=</m:mo>
        <m:mi>m</m:mi>
        <m:msup>
          <m:mi>c</m:mi>
          <m:mn>2</m:mn>
        </m:msup>
      </m:mrow>
    </m:math>
  </formulacontent>
  <formulacaption>
    (<formulanumber detail='formula'>1.1</formulanumber>)
  </formulacaption>
</formula>
```

The caption can also have a label in front of the number. The best way to deal with this still under consideration. I leave it to the reader to wonder how we get the caption at the same level as the content while in practice the number is part of the formula.

Anyway, the previous pages have demonstrated that with version 0.22 of the exporter we can already get a quite acceptable math export. Of course more will follow.

20 E-books: Old wine in new bottles

20.1 Introduction

When Dave Walden asked me if TEX (or CONTEXT) can generate ebooks we exchanged a bit of mail on the topic. Although I had promised myself never to fall into the trap of making examples for the sake of proving something I decided to pick up an experiment that I had been doing with a manual in progress and look into the HTML side of that story. After all, occasionally on the CONTEXT list similar questions are asked, like "Can CONTEXT produce HTML?".[26]

20.2 Nothing new

When you look at what nowadays is presented as an ebook document, there is not much new going on. Of course there are very advanced and interactive documents, using techniques only possible with recent hardware and programs, but the average ebook is pretty basic. This is no surprise. When you take a novel, apart from maybe a cover or an occasional special formatting of section titles, the typesetting of the content is pretty straightforward. In fact, given that formatters like TEX have been around that can do such jobs without much intervention, it takes quite some effort to get that job done badly. It was a bit shocking to notice that on one of the first e-ink devices that became available the viewing was quite good, but the help document was just some word processor output turned into bad-looking PDF. The availability of proper hardware does not automatically trigger proper usage.

I can come up with several reasons why a novel published as an ebook does not look much more interesting and in many cases looks worse. First of all it has to be produced cheaply, because there is also a printed version and because the vendor of some devices also want to make money on it (or even lock you into their technology or shop). Then, it has to be rendered on various devices so the least sophisticated one sets the standard. As soon as it gets rendered, the resolution is much worse than what can be achieved in print, although nowadays I've seen publishers go for quick and dirty printing, especially for reprints.

Over a decade ago, we did some experiments with touch screen computers. They had a miserable battery life, a slow processor and not much memory, but the resolution was the same as on the now fashionable devices. They were

[26] This text appeared in the EUROTEX 2011 proceedings and TUGBoat 101. Thanks to Karl Berry for correcting it.

quite suitable for reading but even in environments where that made sense (for instance to replace carrying around huge manuals), such devices never took off. Nowadays we have wireless access and USB sticks and memory cards to move files around, which helps a lot. And getting a quality comparable to what can be done today was no big deal, at least from the formatting point of view.

In the ConTeXt distribution you will find several presentation styles that can serve as bases for an ebook style. Also some of the ConTeXt manuals come with two versions: one for printing and one for viewing on the screen. A nice example is the MetaFun manual (see figure 20.1) where each page has a different look.

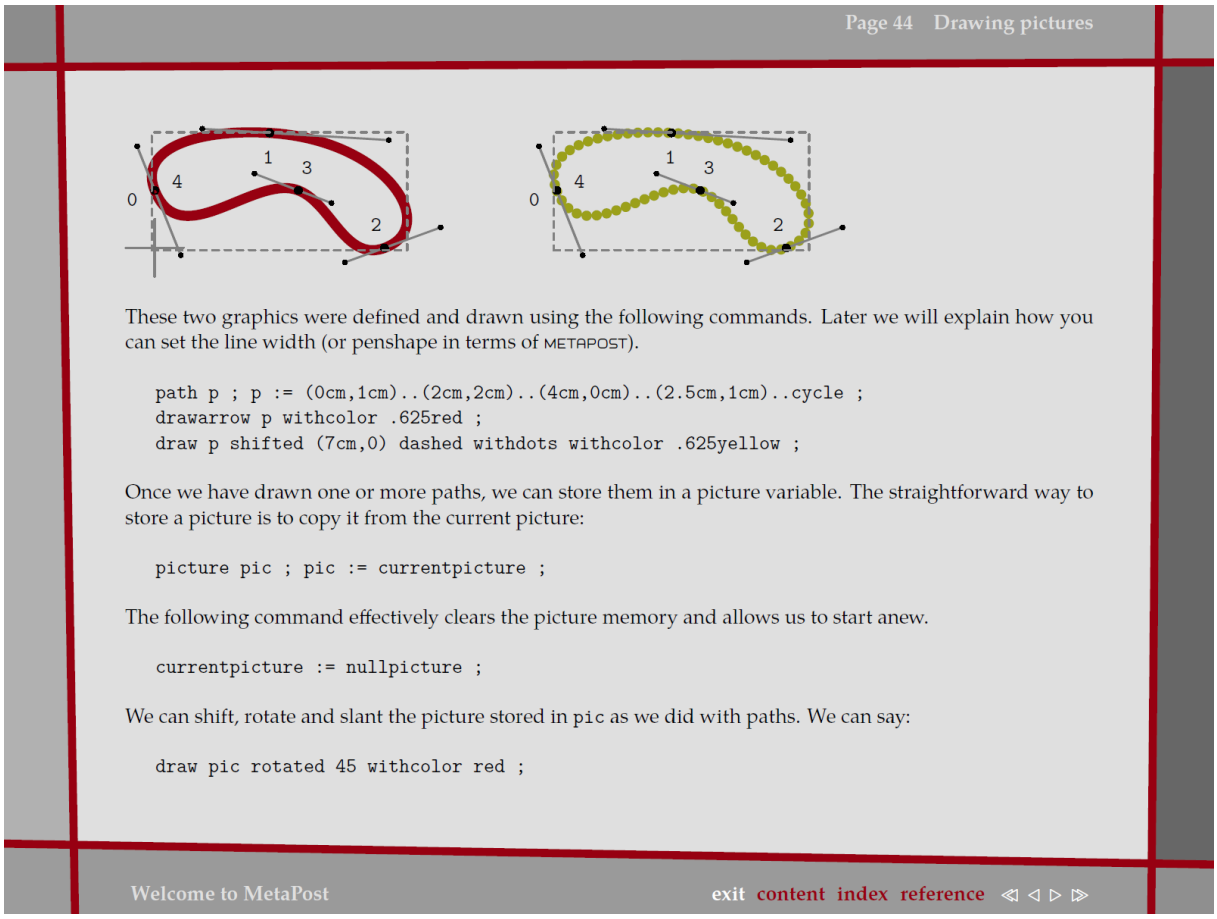


Figure 20.1 A page from the MetaFun manual.

It must be said that the (currently only black and white) devices that use electronic ink have a perceived resolution that is higher than their specifications, due to the semi-analog way the 'ink' behaves. In a similar fashion clever anti-aliasing can do wonders on LCD screens. On the other hand they are somewhat slow and a display refresh is not that convenient. Their liquid crystal counterparts are much faster but they can be tiresome to look at for a long time and reading a book on it sitting in the sun is a no-go. Eventually we will get there and I'm really looking forward to seeing the first device that will use

a high resolution electrowetting CMYK display.[27] But no matter what device is used, formatting something for it is not the most complex task at hand.

20.3 Impact

Just as with phones and portable audio devices, the market for tablets and ebook-only devices is evolving rapidly. While writing this, at work I have one ebook device and one tablet. The ebook device is sort of obsolete because the e-ink screen has deteriorated even without using it and it's just too slow to be used for reference manuals. The tablet is nice, but not that suitable for all circumstances: in the sun it is unreadable and at night the backlight is rather harsh. But, as I mentioned in the previous section, I expect this to change.

If we look at the investment, one needs good arguments to buy hardware that is seldom used and after a few years is obsolete. Imagine that a family of four has to buy an ebook device for each member. Add to that the cost of the books and you quickly can end up with a larger budget than for books. Now, imagine that you want to share a book with a friend: will you give him or her the device? It might be that you need a few more devices then. Of course there is also some data management needed: how many copies of a file are allowed to be made and do we need special programs for that? And if no copy can be made, do we end up swapping devices? It is hard to predict how the situation will be in a few years from now, but I'm sure that not everyone can afford this rapid upgrading and redundant device approach.

A friend of mine bought ebook devices for his children but they are back to paper books now because the devices were not kid-proof enough: you can sit on a book but not on an ebook reader.

The more general devices (pads) have similar problems. I was surprised to see that an iPad is a single user device. One can hide some options behind passwords but I'm not sure if parents want children to read their mail, change preferences, install any application they like, etc. This makes pads not that family friendly and suggests that such a personal device has to be bought for each member. In which case it suddenly becomes a real expensive adventure. So, unless the prices drop drastically, pads are not a valid large scale alternative for books yet.

It might sound like I'm not that willing to progress, but that's not true. For instance, I'm already an enthusiastic user of a media player infrastructure.[28]

---

[27] http://www.liquavista.com/files/LQV0905291LL5-15.pdf

[28] The software and hardware was developed by SlimDevices and currently is available as Logitech

The software is public, pretty usable, and has no vendor lock-in. Now, it would make sense to get rid of traditional audio media then, but this is not true. I still buy CDs if only because I then can rip them to a proper lossless audio format (FLAC). The few FLACs that I bought via the Internet were from self-publishing performers. After the download I still got the CDs which was nice because the booklets are among the nicest that I've ever seen.

Of course it makes no sense to scan books for ebook devices so for that we depend on a publishing network. I expect that at some point there will be proper tools for managing your own electronic books and in most cases a simple file server will do. And the most open device with a proper screen will become my favourite. Also, I would not be surprised if ten years from now, many authors will publish themselves in open formats and hopefully users will be honest enough to pay for it. I'm not too optimistic about the latter, if only because I observe that younger family members fetch everything possible from the Internet and don't bother about rights, so we definitely need to educate them. To some extent publishers of content deserve this behaviour because more than I like I find myself in situations where I've paid some 20 euro for a CD only to see that half a year later you can get it for half the price (sometimes it also happens with books).

Given that eventually the abovementioned problems and disadvantages will be dealt with, we can assume that ebooks are here and will stay forever. So let's move on to the next section and discuss their look and feel.

20.4 Interactivity

The nice thing about a paper book is that it is content and interface at the same time. It is clear where it starts and ends and going from one page to another is well standardized. Putting a bookmark in it is easy as you can fall back on any scrap of paper lying around. While reading you know how far you came and how much there is to come. Just as a desktop on a desktop computer does not resemble the average desktop, an ebook is not a book. It is a device that can render content in either a given or more free-form way.

However, an electronic book needs an interface and this is also where at the moment it gets less interesting. Of course the Internet is a great place to wander around and a natural place to look for electronic content. But there are some arguments for buying them at a bookshop, one being that you see a lot of (potentially) new books, often organized in topics in one glance. It's a different way of selecting. I'm not arguing that the Internet is a worse place, but there

Squeezeserver. Incidentally I can use the iPad as an advanced remote control.

is definitely a difference: more aggressive advertisements, unwanted profiling that can narrow what is presented to a few choices.

Would I enter a bookshop if on the display tables there were stacks of (current) ebook devices showing the latest greatest books? I can imagine that at some point we will have ebook devices that have screens that run from edge to edge and then we get back some of the appeal of book designs. It is that kind of future devices that we need to keep in mind when we design electronic documents, especially when after some decades we want them to be as interesting as old books can be. Of course this is only true for documents that carry the look and feel of a certain time and place and many documents are thrown away. Most books have a short lifespan due to the quality of the paper and binding so we should not become too sentimental about the transition to another medium.

Once you're in the process of reading a book not much interfacing is needed. Simple gestures or touching indicated areas on the page are best. For more complex documents the navigation could be part of the design and no screen real estate has to be wasted by the device itself. Recently I visited a school-related exhibition and I was puzzled by the fact that on an electronic school-board so much space was wasted on colorful nonsense. Taking some 20% off each side of such a device brings down the effective resolution to 600 pixels so we end up with 10 pixels or less per character (shown at about 1 cm width). At the same exhibition there were a lot of compensation programs for dyslexia advertised, and there might be a relationship.

20.5 Formatting

So how important is the formatting? Do we prefer reflow on demand or is a more frozen design that suits the content and expresses the wish of the author more appropriate? In the first case HTML is a logical choice, and in the second one PDF makes sense. You design a nice HTML document but at some point the reflow gets in the way. And yes, you can reflow a PDF file but it's mostly a joke. Alternatively one can provide both which is rather trivial when the source code is encoded in a systematic way so that multiple output is a valid option. Again, this is not new and mostly a matter of a publisher's policy. It won't cost more to store in neutral formats and it has already been done cheaply for a long time.

Somewhat interfering in this matter is digital rights management. While it is rather customary to buy a book and let friends or family read the same book, it can get complicated when content is bound to one (or a few) devices. Not much sharing there, and in the worst case, no way to move your books to a better device. Each year in the Netherlands we have a book fair and bookshops give away a book specially written for the occasion. This year the book was also

available as an ebook, but only via a special code that came with the book. I decided to give it a try and ended up installing a broken application, i.e. I could not get it to load the book from the Internet, and believe me, I have a decent machine and the professional PDF viewer software that was a prerequisite.

20.6 Using TEX

So, back to Dave's question: if CONTEXT can generate ebooks in the Epub format. Equally interesting is the question if TEX can format an Epub file into a (say) PDF file. As with much office software, an Epub file is nothing more than a zip file with a special suffix in which several resources are combined. The layout of the archive is prescribed. However, by demanding that the content itself is in HTML and by providing a stylesheet to control the renderer, we don't automatically get properly tagged and organized content. When I first looked into Epub, I naively assumed that there was some well-defined structure in the content; turns out this is not the case.

Let's start by answering the second question. Yes, CONTEXT can be used to convert an Epub file into a PDF file. The natural followup question is if it can be done automatically, and then some more nuance is needed: it depends. If you download the Epub for "A tale of two cities" from Charles Dickens from the Gutenberg Project website and look into a chapter you will see this:

```
<h1 id="pgepubid00000">A TALE OF TWO CITIES</h1>
<h2 id="pgepubid00001">A STORY OF THE FRENCH REVOLUTION</h2>
<p><br/></p>
<h2>By Charles Dickens</h2>
<p><br/>
<br/></p>
<hr/>
<p><br/>
<br/></p>
<h2 id="pgepubid00002">Contents</h2>
```

What follows is a table of contents formatted using HTML tables and after that

```
<h2 id="pgepubid00004">I. The Period</h2>
```

So, a level two header is used for the subtitle of the book as well as a regular chapter. I must admit that I had to go on the Internet to find this snippet as I wanted to check its location. On my disk I had a similar file from a year ago when I first looked into Epub. There I have:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>I | A Tale of Two Cities</title>
    ....
  </head>
  <body>
    <div class="body">
      <div class="chapter">
        <h3 class="chapter-title">I</h3>
        <h4 class="chapter-subtitle">The Period</h4>
```

I also wanted to make sure if the interesting combination of third and fourth level head usage was still there but it seems that there are several variants available. It is not my intention to criticize the coding, after all it is valid HTML and can be rendered as intended. Nevertheless, the first snippet definitely looks worse, as it uses breaks instead of CSS spacing directives and the second wins on obscurity due to the abuse of the head element.

These examples answer the question about formatting an arbitrary Epub file: "no". We can of course map the tagging to ConTeXt and get pretty good results but we do need to look at the coding.

As such books are rather predictable it makes sense to code them in a more generic way. That way generic stylesheets can be used to render the book directly in a viewer and generic ConTeXt styles can be used to format it differently, e.g. as PDF.

Of course, if I were asked to set up a workflow for formatting ebooks, that would be relatively easy. For instance the Gutenberg books are available as raw text and that can be parsed to some intermediate format or (with MkIV) interpreted directly.

Making a style for a specific instance, like the Dickens book, is not that complex either. After all, the amount of encoding is rather minimal and special bits and pieces like a title page need special design anyway. The zipped file can be processed directly by ConTeXt, but this is mostly just a convenience.

As Epub is just a wrapper, the next question is if ConTeXt can produce some kind of HTML and the answer to that question is positive. Of course this only makes sense when the input is a TeX source, and we have argued before that when multiple output is needed the user might consider a different starting point. After all, ConTeXt can deal with XML directly.

The main advantage of coding in TEX is that the source remains readable and for some documents it's certainly more convenient, like manuals about TEX. In the reference manual 'ConTEXt Lua Documents' (CLD) there are the following commands:

```
\setupbackend
  [export=yes]

\setupinteraction
  [title=Context Lua Documents,
   subtitle=preliminary version,
   author=Hans Hagen]
```

At the cost of at most 10% extra runtime an XML export is generated in addition to the regular PDF file. Given that you have a structured TEX source the exported file will have a decent structure as well and you can therefore transform the file into something else, for instance HTML. But, as we already have a good-looking PDF file, the only reason to have HTML as well is for reflowing. Of course wrapping up the HTML into an Epub structure is not that hard. We can probably even get away from wrapping because we have a single self-contained file.

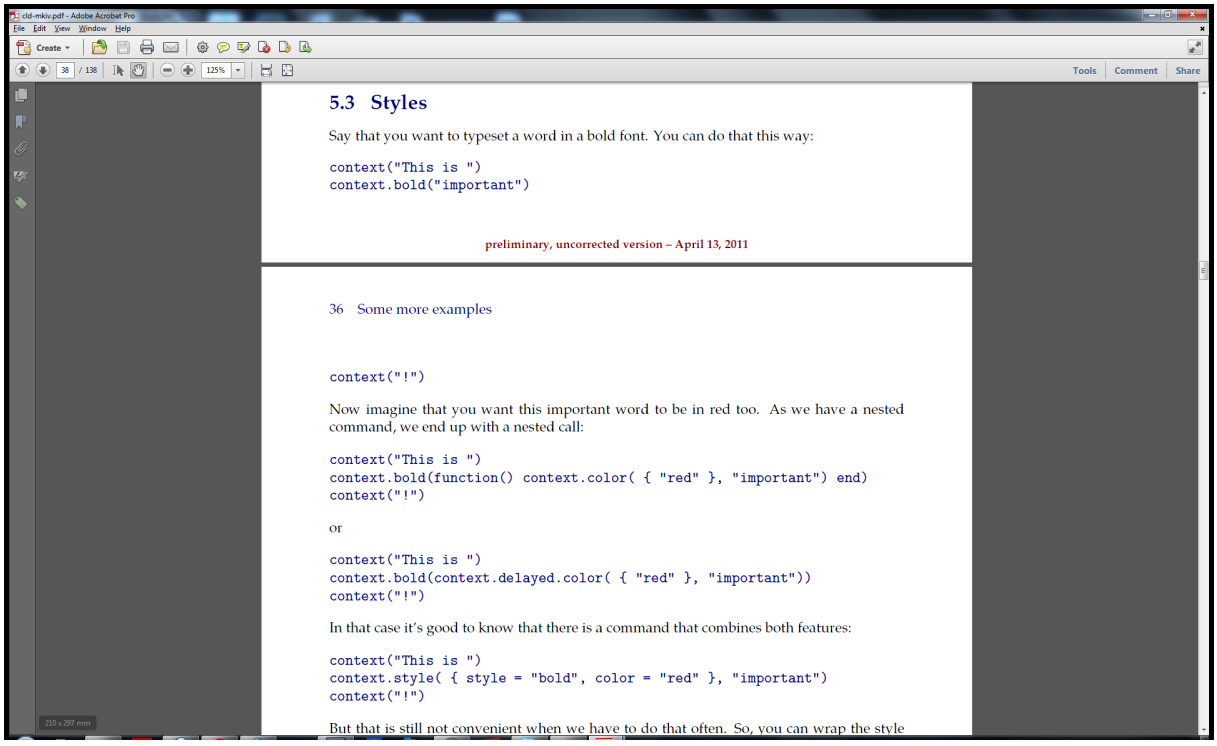


Figure 20.2 A page from the CLD manual in PDF.

The `\setupbackend` command used in the CLD manual has a few more options:

```
\setupbackend
  [export=cld-mkiv-export.xml,
   xhtml=cld-mkiv-export.xhtml,
   css={cld-mkiv-export.css,mathml.css}]
```

We explicitly name the export file and in addition specify a stylesheet and an alternative XHTML file. If you can live without hyperlinks the XML file combined with the cascading style sheet will do a decent job of controlling the formatting.

In the CLD manual chapters are coded like this:

```
\startchapter[title=A bit of Lua]

\startsection[title=The language]
```

The XML output of this

```
<division detail='bodypart'>
  <section detail='chapter' location='aut:3'>
    <sectionnumber>1</sectionnumber>
    <sectiontitle>A bit of Lua</sectiontitle>
    <sectioncontent>
      <section detail='section'>
        <sectionnumber>1.1</sectionnumber>
        <sectiontitle>The language</sectiontitle>
        <sectioncontent>
```

The HTML version has some extra elements:

```
<xhtml:a name="aut_3">
  <section location="aut:3" detail="chapter">
```

The table of contents and cross references have `xhtml:a` elements too but with the `href` attribute. It's interesting to search the web for ways to avoid this, but so far no standardized solution for mapping XML elements onto hyperlinks has been agreed upon. In fact, getting the CSS mapping done was not that much work but arriving at the conclusion that (in 2011) these links could only be done in a robust way using HTML tags took more time.[29] Apart from this the CSS has enough on board to map the export onto something presentable. For instance:

[29] In this example we see the reference `aut:3` turned into `aut_1`. This is done because some browsers like to interpret this colon as a url.

```
sectioncontent {
    display: block ;
    margin-top: 1em ;
    margin-bottom: 1em ;
}

section[detail=chapter], section[detail=title] {
    margin-top: 3em ;
    margin-bottom: 2em ;
}

section[detail=chapter]>sectionnumber {
    display: inline-block ;
    margin-right: 1em ;
    font-size: 3em ;
    font-weight: bold ;
}
```

As always, dealing with verbatim is somewhat special. The following code does the trick:

```
verbatimblock {
    background-color: #9999FF ;
    display: block ;
    padding: 1em ;
    margin-bottom: 1em ;
    margin-top: 1em ;
    font-family: "Lucida Console", "DejaVu Sans Mono", monospace ;
}

verbatimline {
    display: block ;
    white-space: pre-wrap ;
}

verbatim {
    white-space: pre-wrap ;
    color: #666600 ;
    font-family: "Lucida Console", "DejaVu Sans Mono", monospace ;
}
```

The spacing before the first and after the last one differs from the spacing between lines, so we need some extra directives:

```
verbatimlines+verbatimlines {
    display: block ;
    margin-top: 1em ;
}
```

This will format code like the following with a bluish background and inline verbatim with its complement:

```
<verbatimblock detail='typing'>
  <verbatimlines>
    <verbatimline>function sum(a,b)</verbatimline>
    <verbatimline>  print(a, b, a + b)</verbatimline>
    <verbatimline>end</verbatimline>
  </verbatimlines>
</verbatimblock>
```

The hyperlinks need some attention. We need to make sure that only the links and not the anchors get special formatting. After some experimenting I arrived at this:

```
a[href] {
    text-decoration: none ;
    color: inherit ;
}

a[href]:hover {
    color: #770000 ;
    text-decoration: underline ;
}
```

Tables are relatively easy to control. We have tabulate (nicer for text) and natural tables (similar to the HTML model). Both get mapped into HTML tables with css directives. There is some detail available so we see things like this:

```
tablecell[align=flushleft] {
    display: table-cell ;
    text-align: left ;
    padding: .1em ;
}
```

It is not hard to support more variants or detail in the export but that will probably only happen when I find a good reason (a project), have some personal need, or when a user asks for it. For instance images will need some special

attention (conversion, etc.). Also, because we use MetaPost all over the place that needs special care as well, but a regular (novel-like) ebook will not have such resources.

Figure 20.3 A page from CLD manual in HTML.

As an extra, a template file is generated that mentions all elements used, like this:

```
section[detail=summary] {
    display: block ;
}
```

with the inline and display properties already filled in. That way I could see that I still had to add a couple of directives to the final css file. It also became clear that in the CLD manual some math is used that gets tagged as MATHML, so that needs to be covered as well.[30] Here we need to make some decisions as

we export UNICODE and need to consider support for less sophisticated fonts. On the other hand, the W3C consortium has published css for this purpose so we can use these as a starting point. It might be that eventually more tuning will be delegated to the XHTML variant. This is not much extra work as we have the (then intermediate) XML tree available. Thinking of it, we could eventually end up with some kind of css support in CONTEXT itself.

It will take some experimenting and feedback from users to get the export right, especially to provide a convenient way to produce so-called Epub files directly. There is already some support for this container format. If you have enabled XHTML export, you can produce an Epub archive afterwards with:

```
mtxrun --script epub yourfile
```

For testing the results, open source programs like `calibre` are quite useful. It will probably take a while to figure out to what extent we need to support formats like Epub, if only because such formats are adapted on a regular basis.

20.7 The future

It is hard to predict the future. I can imagine that given the user interface that has evolved over ages paper books will not disappear soon. Probably there will be a distinction between read-once and throw-away books and those that you carry with you your whole life as visible proof of that life. I can also imagine that (if only for environmental reasons) ebooks (hopefully with stable devices) will dominate. In that case traditional bookshops will disappear and with them the need for publishers that supply them. Self-publishing will then be most interesting for authors and maybe some of them (or their helpful friends) will be charmed by TEX and tinkering with the layout using the macro language. I can also imagine that at some point new media (and I don't consider an ebook a new medium) will dominate. And how about science fiction becoming true: downloading stories and information directly into our brains.

It reminds me of something I need to do some day soon: get rid of old journals that I planned to read but never will. I would love to keep them electronically but it is quite unlikely that they are available and if so, it's unlikely that I want to pay for them again. This is typically an area where I'd consider using an ebook device, even if it's suboptimal. On the other hand, I don't consider dropping my newspaper subscription yet as I don't see a replacement for the regular coffeestop at the table where it sits and where we discuss the latest

[30] Some more advanced MATHML output will be available when the matrix-related core commands have been upgraded to MKIV and extended to suit today's needs.

news.

The nice thing about an analogue camera is that the image carrier has been standardized and you can buy batteries everywhere. Compare this with their digital cousins: all have different batteries, there are all kinds of memory cards, and only recently has some standardization in lenses shown up. There is a wide range of resolutions and aspect ratios. Other examples of standardization are nuts and bolts used in cars, although it took some time for the metric system to catch on. Books have different dimensions but it's not hard to deal with that property. Where desktop hardware is rather uniform everything portable is different. For some brands you need a special toolkit with every new device. Batteries cannot be exchanged and there are quite some data carriers. On the other hand, we're dealing with software and if we want we can support data formats forever. The Microsoft operating systems have demonstrated that programs written years ago can still run on updates. In addition linux demonstrates that users can take and keep control and create an independence from vendors. So, given that we can still read document sources and given that they are well structured, we can create history-proof solutions. I don't expect that the traditional publishers will play a big role in this if only because of their short term agendas and because changing ownerships works against long term views. And programs like TEX have already demonstrated having a long life span, although it must be said that in today's rapid upgrade cycles it takes some courage to stay with it and its descendants. But downward compatibility is high on the agenda of its users and user groups which is good in the perspective of discussing stable ebooks.

Let's finish with an observation. Books often make a nice (birthday) present and finding one that suits is part of the gift. Currently a visible book has some advantages: when unwrapped it can be looked at and passed around. It also can be a topic of discussion and it has a visible personal touch. I'm not so sure if vouchers for an ebook have the same properties. It probably feels a bit like giving synthetic flowers. I don't know what percentage of books is given as presents but this aspect cannot be neglected. Anyway, I wonder when I will buy my first ebook and for who. Before that happens I'll probably have generated lots of them.

21 Italic correction

21.1 Introduction

During the 2011 ConTeXt conference there were presentations by Thomas Schmitz and Jano Kula where they demonstrated advanced rendering of document source encoded in xml. When looking at the examples on screen using many fonts I realized that (also given my own workflows) it was about time to look into automated italic correction in the perspective of MkIV.

In the Lucida Math project it already became clear that italics in OpenType math fonts are to be ignored. And, as in regular OpenType fonts italic correction is basically non-existent some alternative approach is needed there as well. In ConTeXt you can already for quite a while enable the `itlc` feature which adds italic correction to shapes using some heuristics. However, in TeX this kind of correction is never applied automatically but is triggered by the `\/` command. Commands like `\em` deal with italic correction automatically but otherwise you need to take care of it yourself. In a time when you not always have control over the source code or when you are coding in a format that has no provisions for it (for instance xml) some further automatism makes sense. You might even wonder if explicit corrections still make sense.

In this chapter we discuss an alternative approach in MkIV. This is a typical example of an experimental feature that might need further discussion (probably at a next conference). One of our mottos is that the document source should be as clean as possible and this is one way to go.

21.2 Some preparations

Adding italic correction to a font is easy: you just add the right feature directive. You can do this for all italic (or oblique) fonts in one go:

```
\definefontfeature[default][default][itlc=yes]
```

At some point this might become the default in ConTeXt. After that the `\/` command can do the job, but as mentioned, we don't really want to do this each time it's needed. If you never plan to use that command you can disable TeX's built-in mechanism completely by setting the `textitalics` parameter.

```
\definefontfeature[default][default][itlc=yes,textitalics=yes]
```

It even makes sense then to redefine the the `\/` command:

```
\let\/=/
```

so that we have a nice escape in tune with the other escapes.

21.3 Controlling correction

In the following examples we will use Cambria as an example as it shows the effect rather prominently.

We start with a simple case: just an emphasized word in a small line:

```
\setupitaliccorrection[none]\tf test {\it test} test
\setupitaliccorrection[none]\tf test {\it test\/} test}
\setupitaliccorrection[text]\tf test {\it test} test}
```

Decorated for the purpose of this demonstration this comes out as follows:

In the first line no correction is applied. The second line shows TEX in action and the third line demonstrates the automatically applied correction. The explicit directive in the second lines of course gives most control but is also a no-go when you have lots of them.

Actually, TEX is clever enough to ignore multiple corrections: it will only apply one after a glyph.

```
\setupitaliccorrection[none]\tf test {\it test} test}
\setupitaliccorrection[none]\tf test {\it test\/} test}
\setupitaliccorrection[none]\tf test {\it test\/\/\/\/} test}
```

So we get this:

It can be argued that in a decent usage of ConTeXt you will never switch to another font this way. Instead you will do this:

```
\definehighlight[important][style=\it]

test \important{test} test
```

However, this will not correct at all, so in fact you have to use an environment that takes care of automatically adding the `\/` at the end. Quite from the start the `\em` command does this, with the added benefit of dealing with bold and nested emphasizing.

Which brings us to cases where you don't want to apply correction, like:

```
\setupitaliccorrection[none]\tf test {\it test}{\bi test}
\setupitaliccorrection[none]\tf test {\it test\/}{\bi test}
\setupitaliccorrection[text]\tf test {\it test}{\bi test}
```

Now we get:

A variant on this is:

```
\setupitaliccorrection[none]\tf test {\it test   \bi test}
\setupitaliccorrection[none]\tf test {\it test\/ \bi test}
\setupitaliccorrection[text]\tf test {\it test   \bi test}
```

which gives:

So, no italic correction is added between italic shapes of different fonts. Ideally we should have some inter-character kerning, but that is currently beyond this mechanism.

What does the `text` mean in the setup command? The following table tells what keywords can be passed:

`text`	only apply correction to running text
`always`	also apply correction to end end of a list
`global`	enable this mechanism globally (more efficient)
`none`	disable this mechanism

The difference between `text` and `always` is best demonstrated with an example:

```
\setupitaliccorrection[none]\tf   test {\it test}}
\setupitaliccorrection[always]\tf test {\it test}}
\setupitaliccorrection[text]\tf   test {\it test}}
```

This gives:

The `always` option will flush pending corrections at a boundary, like the edge of a box (or line). Contrary to TEX's italic corrections, the MKIV variants are glue and they will disappear whenever TEX likes to get rid of glue, for instance at line breaks.[31]

[31] There is some room for improvement here, for instance we can take penalties into account.

While writing this, we're still talking of an experimental setup so there might be extensions or changes to this mechanism.[32]

As it's just a guess you can influence the amount of automatic correction by specifying a factor. We show an exmaple of this.

```
\definefontfeature[itclyes]       [default][itlc=yes,textitalics=delay]
\definefontfeature[itclyesten]    [default][itlc=10, textitalics=delay]
\definefontfeature[itclyeshundred][default][itlc=100,textitalics=delay]

\definefont[itlcitalicyes]      [name:cambriaitalic*itclyes        sa 4]
\definefont[itlcitalicten]      [name:cambriaitalic*itclyesten     sa 4]
\definefont[itlcitalichundred]  [name:cambriaitalic*itclyeshundred sa 4]
```

We show all three variants:

```
\setupitaliccorrection[text]\itlcregular test {\itlcitalicyes     test}
test\par
\setupitaliccorrection[text]\itlcregular test {\itlcitalicten     test}
test\par
\setupitaliccorrection[text]\itlcregular test {\itlcitalichundred test}
test\par
```

This becomes:

test *test* test
test *test* test
test *test* test

21.4 Saving resources

You can keep track of what gets done by enabling a tracker:

```
\enabletrackers[typesetters.italics]
```

You will notice that there are occasional reports about correction being inserted,

[32] For instance, I'm considering extending this mechanism to provide kerning between fonts, something for a rainy afternoon.

ignored and removed. As node lists are parsed there is some extra overhead, but not that much. The TEX solution (using \/) is quit efficient because that command directly injects a kern without too much analysis. You can gain some efficiency for the automated variant by using the global option:

```
\setupitaliccorrection[always,global]
```

Also, you can disable TEX's mechanism effectively by not passing the italic information to the font machinery at all:

```
\definefontfeature[italics][default][itlc=yes,textitalics=yes]
```

The itlc feature will tag the font for italic corrections but the textitalics option will make sure that this information is not passed to the TEX font handler but kept private.

As adding the italic corrections to a font takes memory and a little bit of extra load time, we can delay this process till it is really needed.

```
\definefontfeature[italics][default][itlc=yes,textitalics=delay]
```

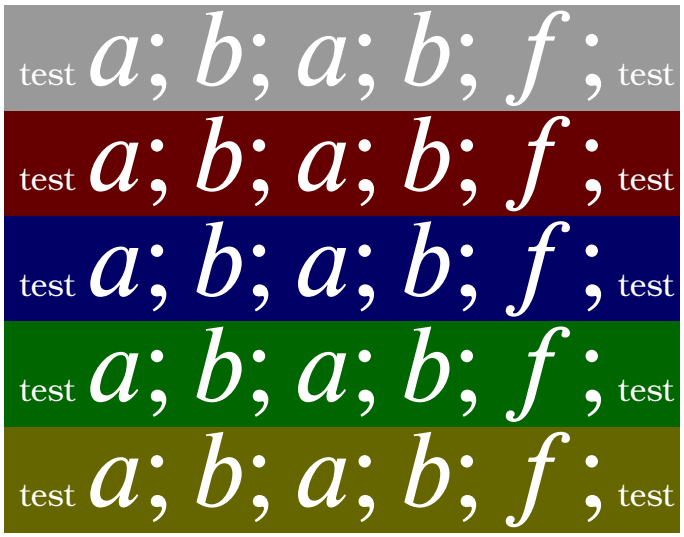
In this case the correction will be calculated when needed and cached for later usage. At some point this might become the default ConTEXt behaviour.

21.5 Math

Italic correction in math plays a role when dealing with traditional TEX fonts, where glyph dimensions can have a special meaning. However, in OpenType math the correction is mostly ignored. You can disable it altogether and let an alternative mechanism deal with it. This mechanism is still somewhat experimental but is controlled as follows:

```
\switchtobodyfont[xits]
\setupmathematics[italics=no] test $a;b;a; b; f;$ test}
\setupmathematics[italics=1]  test $a;b;a; b; f;$ test}
\setupmathematics[italics=2]  test $a;b;a; b; f;$ test}
\setupmathematics[italics=3]  test $a;b;a; b; f;$ test}
\setupmathematics[italics=4]  test $a;b;a; b; f;$ test}
```

This gives:



The actual rendering can depend on the settings in the goodies file, for instance:

```
local italics = {
    defaultfactor = 0.025,
    disableengine = true, % feature: mathitalics=yes
    corrections   = {
     -- [0x1D44E] = 0.99,    -- a (fraction of quad)
     -- [0x1D44F] = 100,     -- b (font points)
        [0x1D453] = -0.0375, -- f
    }
}

return {
    name = "xits-math",
    version = "1.00",
    comment = "Goodies that complement xits (by Khaled Hosny).",
    author = "Hans Hagen",
    copyright = "ConTeXt development team",
    mathematics = {
        italics = {
            ["xits-math"] = italics,
        },
    }
}
```

Corrections can be specified in the font's units or as a fraction (smaller than 1) in which case it will be multiplied by `1em`. You can set the font feature `mathitalics` to `yes` to inhibit the engine's built-in mechanism completely and rely on the alternative approach but as users will seldom define math feature

sets themselves, there is also the possibility to disable the engine in the goodies file.

The process can be watched by setting a tracker:

```
\enabletrackers[math.italics]
```

22 Optical optimization

One of the objectives of the oriental TEX project has always been to play with paragraph optimization. The original assumption was that we needed an advanced non-standard paragraph builder to Arabic done right but in the end we found out that a more straightforward approach is to use a sophisticated OpenType font in combination with a paragraph postprocessor that uses the advanced font capabilities. This solution is somewhat easier to imagine that a complex paragraph builder but still involves quite some juggling.

At the June 2012 meeting of the ntg there was a talk about typesetting Devanagari and as fonts are always a nice topic (if only because there is something to show) it made sense to tell a bit more about optimizing Arabic at the same time. In fact, that presentation was already a few years too late because a couple of years back, when the oriental TEX project was presented at TUG and Dante meetings, the optimizer was already part of the ConTEXt core code. The main reason for not advocating is was the simple fact that no font other than the (not yet finished) Husayni font provided the relevant feature set.

The lack of advanced fonts does not prevent us from showing what we're dealing with. This is because the ConTEXt mechanisms are generic in the sense that they can also be used with regular Latin fonts, although it does not make that much sense. Of course only MkIV is supported. In this chapter we will stick to Latin. A more extensive article is published by Idris Samawi Hamid and myself in the proceedings of the combined euroTEXand ConTEXt conference.

When discussing optical optimization of a paragraph, a few alternatives come to mind:

- One can get rid of extensive spaces by adding additional kerns between glyphs. This is often used by poor mans typesetting programns (or routines) and can be applied to non-connecting scripts. It just looks bad.

- Glyphs can be widened a few percent and this is an option that LuaTEX inherits from its predecessor pdfTEX. Normally this goes unnoticed although excessive scaling makes things worse, and yes, one can run into such examples. This strategy goes under the name hz-optimization (the hz refers to Hermann Zaph, who first came with this solution).

- A real nice solution is to replace glyphs by narrower or wider variants. This is in fact the ideal hz solution but for it to happen one not only needs needs fonts with alternative shapes, but also a machinery that can deal with them.

- An already old variant is the one first used by Gutenberg, who used alternative cuts for certain combinations of characters. This is comparable with ligatures. However, to make the look and feel optimal, one needs to analyze the text and make decisions on what to replace without loosing consistency.

The solution described here does a bit of everything. As it is mostly meant for a connective script, the starting point is how a scribe works when filling up a line nicely. Depending on how well he or she can see it coming, the writing can be adapted to widen or narrow following words. And it happens that in Arabic scripts there are quite some ways to squeeze more characters in a small area and/or expand some to the extreme to fill up the available space. Shapes can be wider or narrower, they can be stacked and they can get replaced by ligatures. Of course there is some interference with the optional marks on top and below but even there we have some freedom. The only condition is that the characters in a word stay connected.

So, given enough alternative glyphs, one can imagine that excessive interword spacing can be avoided. However, it is non-trivial to check all possible combinations. Actually, it is not needed either, as esthetic rules put some bounds on what can be done. One should more think in terms of alternative strategies or solutions and this is the terminology that we will therefore use.

Easiest is to demonstrate this with Latin, if only because it's more intuitive to see what happens. This is not the place to discuss all the gory details so you have to take some of the configuration options on face value. Once this mechanism is stable and used, the options can be described. For now we stick to presenting the idea.

Let's assume that you know what font features are. The idea is to work with combinations of such features and figure out what combination suits best. In order not to clutter a document style, these sets are defined in so called goodie files. Here is an except of `demo.lfg`:

```
return {
  name = "demo",
  version = "1.01",
  comment = "An example of goodies.",
  author = "Hans Hagen",
  featuresets = {
    simple = {
      mode   = "node",
      script = "latn"
    },
    default = {
```

```
      mode   = "node",
      script = "latn",
      kern   = "yes",
    },
    ligatures = {
      mode   = "node",
      script = "latn",
      kern   = "yes",
      liga   = "yes",
    },
    smallcaps = {
      mode   = "node",
      script = "latn",
      kern   = "yes",
      smcp   = "yes",
    },
  },
  solutions = {
    experimental = {
      less = {
        "ligatures", "simple",
      },
      more = {
        "smallcaps",
      },
    },
  },
}
```

We see four sets of features here. You can use these sets in a ConTeXt feature definition, like:

```
\definefontfeature
  [solution-demo]
  [goodies=demo,
   featureset=default]
```

You can use a set as follows:

```
\definefont
  [SomeTestFont]
  [texgyrepagellaregular*solution-demo at 10pt]
```

So far, there is nothing special and new, but we can go a step further.

```
\definefontsolution
  [solution-a]
  [goodies=demo,
   solution=experimental,
   method={normal,preroll},
   criterium=1]

\definefontsolution
  [solution-b]
  [goodies=demo,
   solution=experimental,
   method={normal,preroll,split},
   criterium=1]
```

Here we have defined two solutions. They refer to the `experimental` solution in the goodie file `demo.lfg`. A solution has a `less` and a `more` entry. The featuresets mentioned there reflect ways to make a word narrower of wider. There can be more than one way to do that, although it comes at a performance price. Before we see how this works out we turn on a tracing option:

```
\enabletrackers
  [builders.paragraphs.solutions.splitters.colors]
```

This will color the words in the result according to what has happened. When a featureset out of the `more` category has been applied, the words turn green, when `less` is applied, the word becomes yellow. The `preroll` option in the `method` list makes sure that we do a more extensive test beforehand.

```
\SomeTestFont \startfontsolution[solution-a]
\input zapf \par
\stopfontsolution
```

In figure 22.1 we see what happens. In each already split line words get wider or narrower until we're satisfied. A criterium of 1 is pretty strict[33]. Keep in mind that we use some arbitrary features here. We try removing kerns to get narrower although there is nothing that guarantees that kerns are positive. On the other hand, using ligatures might help. In order to get wider we use smallcaps. Okay, the result will look somewhat strange but so does much typesetting nowadays.

[33] This number reflects the maximum badness and future versions might have a different measure with more granularity.

There is one pitfall here. This mechanism is made for a connective script where hyphenation is not used. As a result a word here is actually split up when it has discretionaries and of course this text fragment has. It goes unnoticed in the rendering but is of course far from optimal.

```
\SomeTestFont \startfontsolution[solution-b]
\input zapf \par
\stopfontsolution
```

In this example (figure 22.2) we keep words as a whole but as a side effect we skip words that are broken across a line. This is mostly because it makes not much sense to implement it as Latin is not our target. Future versions of ConTeXt might get more sophisticated font machinery so then things might look better.

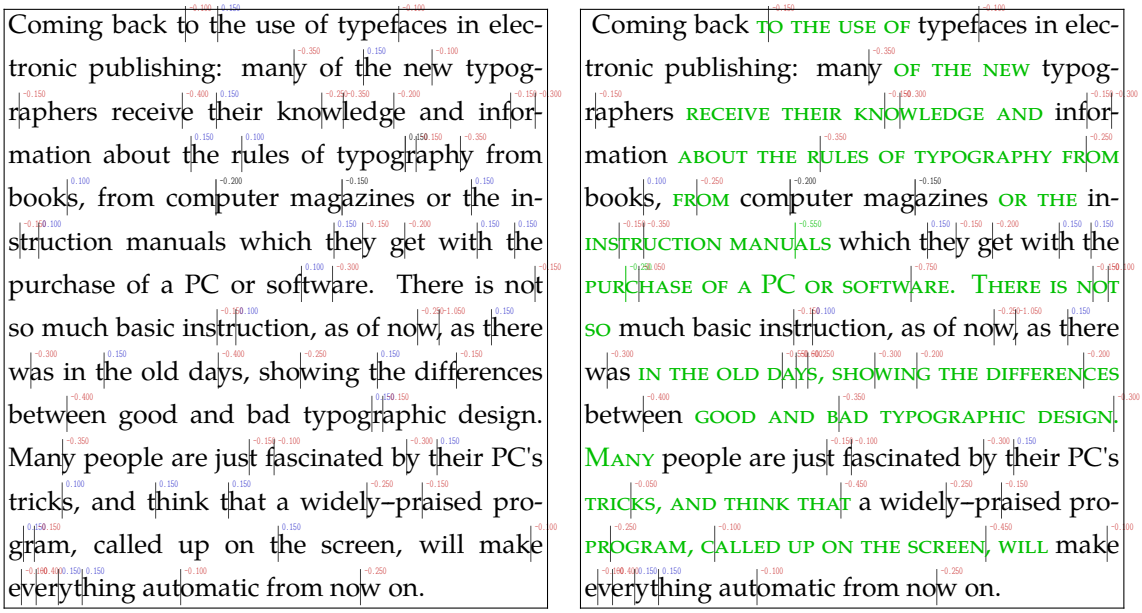
We show two more methods:

```
\definefontsolution
  [solution-c]
  [goodies=demo,
   solution=experimental,
   method={reverse,preroll},
   criterium=1]

\definefontsolution
  [solution-d]
  [goodies=demo,
   solution=experimental,
   method={random,preroll,split},
   criterium=1]
```

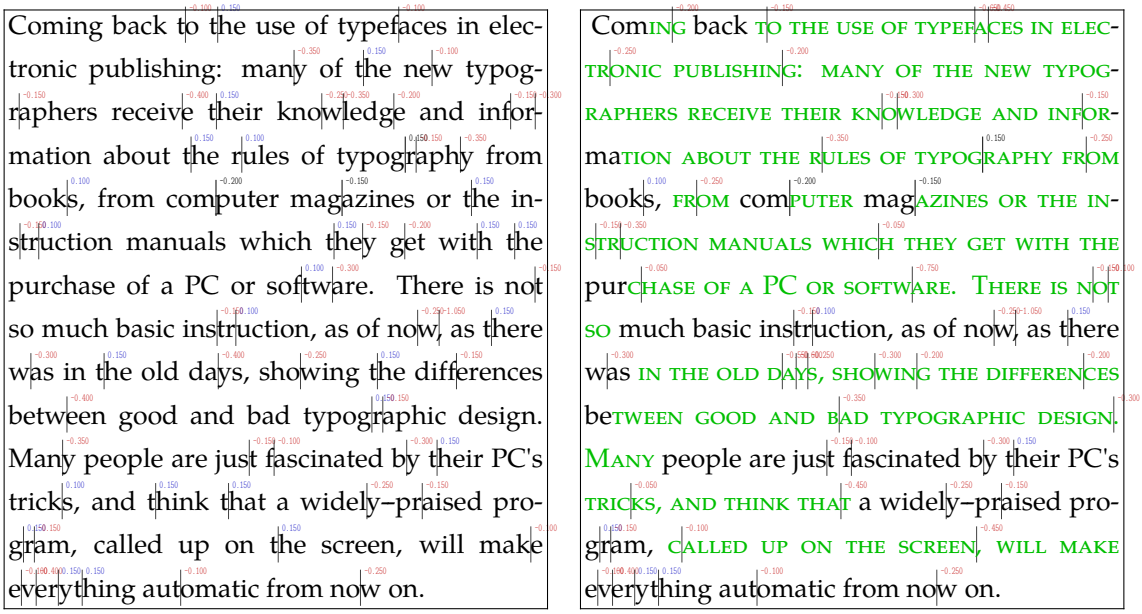
In figure 22.3 we start at the other end of a line. As we sort of mimick a scribe, we can be one who plays safe at the start of corrects at the end.

In figure 22.4 we add some randomness but to what extent this works well depends on how many words we need to retypeset before we get the badness of the line within the constraints.



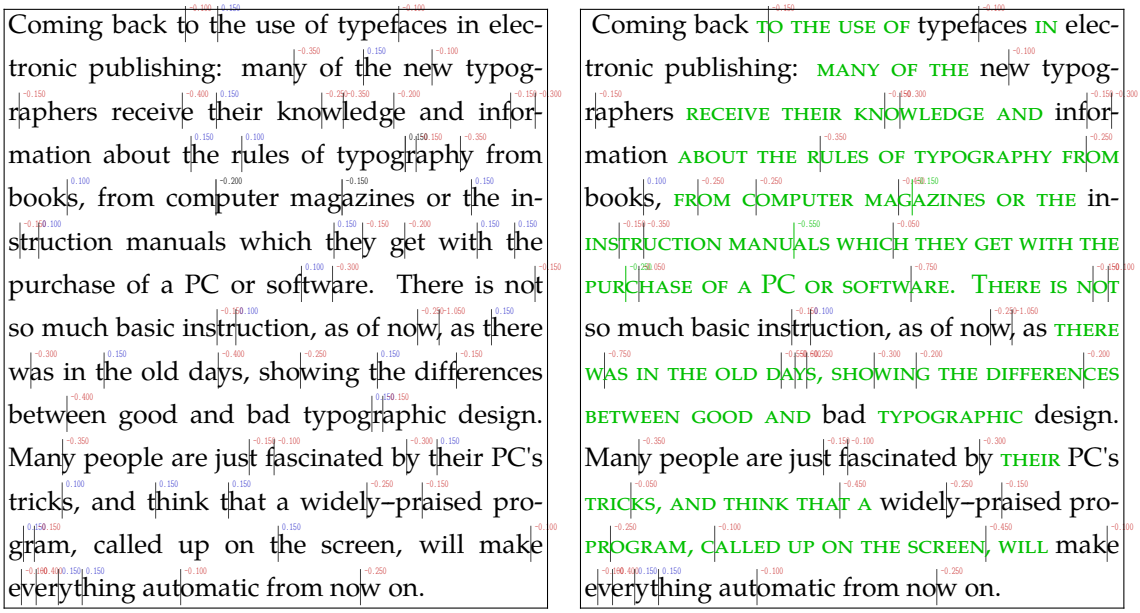
normal solution

Figure 22.1 Solution a.



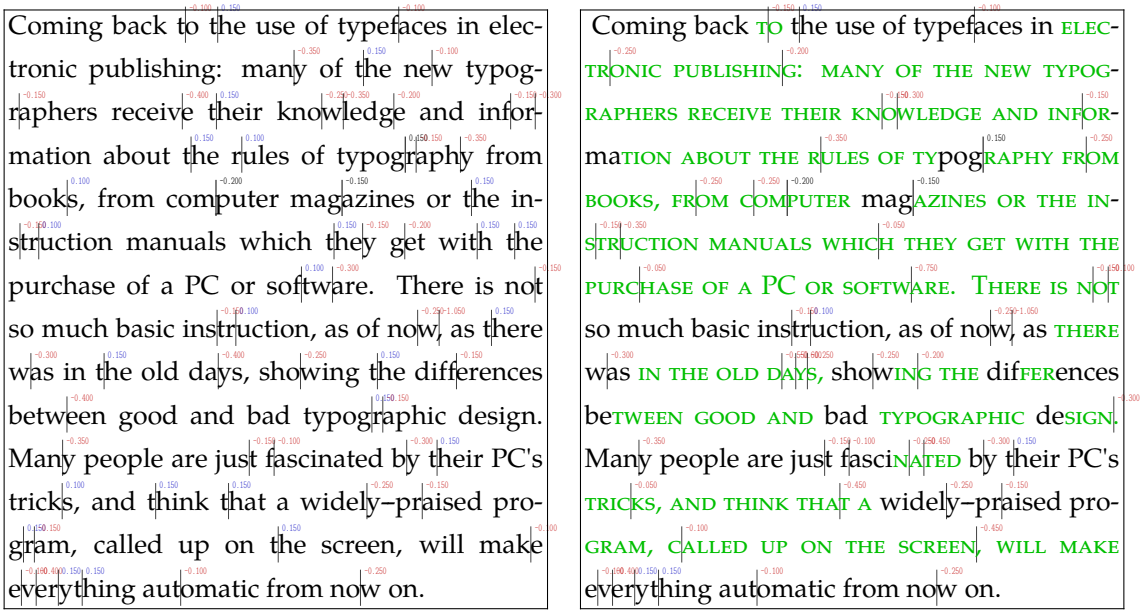
normal solution

Figure 22.2 Solution b.



normal solution

Figure 22.3 Solution c.



normal solution

Figure 22.4 Solution d.

23 Updating the code base

23.1 Introduction

After much experimenting with new code in MkIV a new stage in ConTeXt development was entered in the last quarter of 2011. This was triggered by several more or less independent developments. I will discuss some of them here since they are a nice illustration of how ConTeXt evolves. This chapter was published in TugBoat 103; thanks to Karl Berry and Barbara Beeton for making it better.

23.2 Interfacing

Wolfgang Schuster, Aditya Mahajan and I were experimenting with an abstraction layer for module writers. In fact this layer itself was a variant of some new mechanisms used in the MkIV structure related code. That code was among the first to be adapted as it is accompanied by much Lua code and has been performing rather well for some years now.

In ConTeXt most of the user interface is rather similar and module writers are supposed to follow the same route as the core of ConTeXt. For those who have looked in the source the following code might look familiar:

```
\unexpanded\def\mysetupcommand
  {\dosingleempty\domysetupcommand}

\def\domysetupcommand[#1]%
  {..........
   \getparameters[\??my][#1]%
   ..........
   ..........}
```

This implements the command `\mysetupcommand` that is used as follows:

```
\mysetupcommand[color=red,style=bold,...]
```

The above definition uses three rather low-level interfacing commands. The `\unexpanded` makes sure that the command does not expand in unexpected ways in cases where expansion is less desirable. (Aside: The ConTeXt `\unexpanded` prefix has a long history and originally resulted in the indirect definition of a macro. That way the macro could be part of testing (expanded) equivalence. When ε-TeX functionality showed up we could use `\protected` but we stuck to

the name \unexpanded. So, currently ConTeXt's \unexpanded is equivalent to ε-TeX's \protected. Furthermore, in ConTeXt \expanded is not the same as the ε-TeX primitive. In order to use the primitives you need to use their \normal... synonyms.) The \dosingleempty makes sure that one argument gets seen by injecting a dummy when needed. At some point the \getparameters command will store the values of keys in a namespace that is determined by \??my. The namespace used here is actually one of the internal namespaces which can be deduced from the double question marks. Module namespaces have four question marks.

There is some magic involved in storing the values. For instance, keys are translated from the interface language into the internal language which happens to be English. This translation is needed because a new command is generated:

```
\def\@@mycolor{red}
\def\@@mystyle{bold}
```

and such a command can be used internally because in so-called unprotected mode @?! are valid in names. The Dutch equivalent is:

```
\mijnsetupcommando[kleur=rood,letter=vet]
```

and here the kleur has to be converted into color before the macro is constructed. Of course values themselves can stay as they are as long as checking them uses the internal symbolic names that have the language specific meaning.

```
\c!style{color}
\k!style{kleur}
\v!bold {vet}
```

Internally assignments are done with the \c! variant, translation of the key is done using the \k! alternative and values are prefixed by \v!.

It will be clear that for the English user interface no translation is needed and as a result that interface is somewhat faster. There we only need

```
\c!style{color}
\v!bold {bold}
```

Users never see these prefixed versions, unless they want to define an internationalized style, in which case the form

```
\mysetupcommand[\c!style=\v!bold]
```

has to be used, as it will adapt itself to the user interface. This leaves the `\??my` that in fact expands to `\@@my`. This is the namespace prefix.

Is this the whole story? Of course it isn't, as in ConTeXt we often have a generic instance from which we can clone specific alternatives; in practice, the `\@@mycolor` variant is used in a few cases only. In that case a setup command can look like:

```
\mysetupcommand[myinstance][style=bold]
```

And access to the parameters is done with:

```
\getvalue{\??my myinstance\c!color}
```

So far the description holds for MkII as well as MkIV, but in MkIV we are moving to a variant of this. At the cost of a bit more runtime and helper macros, we can get cleaner low-level code. The magic word here is `commandhandler`. At some point the new MkIV code started using an extra abstraction layer, but the code needed looked rather repetitive despite subtle differences. Then Wolfgang suggested that we should wrap part of that functionality in a definition macro that could be used to define module setup and definition code in one go, thereby providing a level of abstraction that hides some nasty details. The main reason why code could look cleaner is that the experimental core code provided a nicer inheritance model for derived instances and Wolfgang's letter module uses that extensively. After doing some performance tests with the code we decided that indeed such an initializer made sense. Of course, after that we played with it, some more tricks were added, and eventually I decided to replace the similar code in the core as well, that is: use the installer instead of defining helpers locally.

So, how does one install a new setup mechanism? We stick to the core code and leave modules aside for the moment.

```
\definesystemvariable{my}

\installcommandhandler \??my {whatever} \??my
```

After this command we have available some new helper commands of which only a few are mentioned here (after all, this mechanism is still somewhat experimental):

```
\setupwhatever[key=value]
\setupwhatever[instance][key=value]
```

Now a value is fetched using a helper:

```
\namedwhateverparameter{instance}{key}
```

However, more interesting is this one:

```
\whateverparameter{key}
```

For this to work, we need to set the instance:

```
\def\currentwhatever{instance}
```

Such a current state macro already was used in many places, so it fits into the existing code quite well. In addition to `\setupwhatever` and friends, another command becomes available:

```
\definewhatever[instance]
\definewhatever[instance][key=value]
```

Again, this is not so much a revolution as we can define such a command easily with helpers, but it pairs nicely with the setup command. One of the goodies is that it provides the following feature for free:

```
\definewhatever[instance][otherinstance]
\definewhatever[instance][otherinstance][key=value]
```

In some cases this creates more overhead than needed because not all commands have instances. On the other hand, some commands that didn't have instances yet, now suddenly have them. For cases where this is not needed, we provide simple variants of commandhandlers.

Additional commands can be hooked into a setup or definition so that for instance the current situation can be updated or extra commands can be defined for this instance, such as `\start...` and `\stop...` commands.

It should be stressed that the installer itself is not that special in the sense that we could do without it, but it saves some coding. More important is that we no longer have the `@@` prefixed containers but use `\whateverparameter` commands instead. This is definitely slower than the direct macro, but as we often deal with instances, it's not that much slower than `\getvalue` and critical components are rather well speed-optimized anyway.

There is, however, a slowdown due to the way inheritance is implemented. That

is how this started out: using a different (but mostly compatible) inheritance model. In the MkII approach (which is okay in itself) inheritance happens by letting values point to the parent value. In the new model we have a more dynamic chain. It saves us macros but can expand quite wildly depending on the depth of inheritance. For instance, in sectioning there can easily be five or more levels of inheritance. So, there we get slower processing. The same is true for `\framed` which is a rather critical command, but there it is nicely compensated by less copying. My personal impression is that due to the way ConTeXt is set up, the new mechanism is actually more efficient on an average job. Also, because many constructs also depend on the `\framed` command, that one can easily be part of the chain, which again speeds up a bit. In any case, the new mechanisms use much less hash space.

Some mechanisms still look too complex, especially when they hook into others. Multiple inheritance is not trivial to deal with, not only because the meaning of keys can clash, but also because supporting it would demand quite complex fully expandable resolvers. So for the moment we stay away from it. In case you wonder why we cannot delegate more to Lua: it's close to impossible to deal with TeX's grouping in efficient ways at the Lua end, and without grouping available TeX becomes less useful.

Back to the namespace. We already had a special one for modules but after many years of ConTeXt development, we started to run out of two character combinations and many of them had no relation to what name they spaced. As the code base is being overhauled anyway, it makes sense to also provide a new core namespace mechanism. Again, this is nothing revolutionary but it reads much more nicely.

```
\installcorenamespace {whatever}

\installcommandhandler \??whatever {whatever} \??whatever
```

This time deep down no `@@` is used, but rather something more obscure. In any case, no one will use the meaning of the namespace variables, as all access to parameters happens indirectly. And of course there is no speed penalty involved; in fact, we are more efficient. One reason is that we often used the prefix as follows:

```
\setvalue{\??my:option:bla}{foo}
```

and now we just say:

```
\installcorenamespace {whateveroption}
```

```
\setvalue{\??whateveroption bla}{foo}
```

The commandhandler does such assignments slightly differently as it has to prevent clashes between instances and keywords. A nice example of such a clash is this:

```
\setvalue{\??whateveroption sectionnumber}{yes}
```

In sectioning we have instances named `section`, but we also have keys named `number` and `sectionnumber`. So, we end up with something like this:

```
\setvalue{\??whateveroption section:sectionnumber}{yes}
\setvalue{\??whateveroption section:number}{yes}
\setvalue{\??whateveroption :number}{yes}
```

When I decided to replace code similar to that generated by the installer a new rewrite stage was entered. Therefore one reason for explaining this here is that in the process of adapting the core code instabilities are introduced and as most users use the beta version of MkIV, some tolerance and flexibility is needed and it might help to know why something suddenly fails.

In itself using the commandhandler is not that problematic, but wherever I decide to use it, I also clean up the related code and that is where the typos creep in. Fortunately Wolfgang keeps an eye on the changes so problems that users report on the mailing lists are nailed down relatively fast. Anyway, the rewrite itself is triggered by another event but that one is discussed in the next section.

We don't backport (low-level) improvements and speedups to MkII, because for what we need TEX for, we consider pdfTEX and XETEX rather obsolete. Recent tests show that at the moment of this writing a LuaTEX MkIV run is often faster than a comparable pdfTEX MkII run (using utf-8 and complex font setups). When compared to a XETEX MkII run, a LuaTEX MkIV run is often faster, but it's hard to compare, as we have advanced functionality in MkIV that is not (or differently) available in MkII.

23.3 Lexing

The editor that I use, called SciTE, has recently been extended with an extra external lexer module that makes more advanced syntax highlighting possible, using the Lua lpeg library. It is no secret that the user interface of ConTEXt is also determined by the way structure, definitions and setups can be high-

lighted in an editor.[34] When I changed to SciTE I made sure that we had proper highlighting there.

At Pragma ADE one of the leading principles has always been: if the document source looks bad, mistakes are more easily made and the rendering will also be affected. Or phrased differently: if we cannot make the source look nice, the content is probably not structured that well either. The same is true for TEX source, although to a large extent there one must deal with the specific properties of the language.

So, syntax highlighting, or more impressively: lexing, has always been part of the development of ConTEXt and for instance the pretty printers of verbatim provide similar features. For a long time we assumed line-based lexing, mostly for reasons of speed. And surprisingly, that works out quite well with TEX. We used a simple color scheme suitable for everyday usage, with not too intrusive coloring. Of course we made sure that we had runtime spell checking integrated, and that the different user interfaces were served well.

But then came the lpeg lexer. Suddenly we could do much more advanced highlighting. Once I started playing with it, a new color scheme was set up and more sophisticated lexing was applied. Just to mention a few properties:

- We distinguish between several classes of macro names: primitives, helpers, interfacing, and user macros.
- In addition we highlight constant values and special registers differently.
- Conditional constructs can be recognized and are treated as in any regular language (keep in mind that users can define their own).
- Embedded MetaPost code is lexed independently using a lexer that knows the language's primitives, helpers, user macros, constants and of course specific syntax and drawing operators. Related commands at the TEX end (for defining and processing graphics) are also dealt with.
- Embedded Lua is lexed independently using a lexer that not only deals with the language but also knows a bit about how it is used in ConTEXt. Of course the macros that trigger Lua code are handled.
- Metastructure and metadata related macros are colored in a fashion similar to constants (after all, in a document one will not see any constants, so there is no color clash).
- Some special and often invisible characters get a special background color so that we can see when there are for instance non-breakable spaces sitting there.
- Real-time spell checking is part of the deal and can optionally be turned

[34] It all started with `wdt`, `texedit` and `texwork`, editors and environments written by myself in Modula2 and later in Perl Tk, but that was in a previous century.

on. There we distinguish between unknown words, known but potentially misspelled words, and known words.

Of course we also made lexers for MetaPost, Lua, xml, pdf and text documents so that we have a consistent look and feel.

When writing the new lexer code, and testing it on sources, I automatically started adapting the source to the new lexing where possible. Actually, as cleaning up code is somewhat boring, the new lexer is adding some fun to it. I'm not so sure if I would have started a similar overhaul so easily otherwise, especially because the rewrite now also includes speedup and cleanup. At least it helps to recognize less desirable left-overs of MkII code.

23.4 Hiding

It is interesting to notice that users seldom define commands that clash with low level commands. This is of course a side effect of the fact that one seldom needs to define a command, but nevertheless. Low-level commands were protected by prefixing them by one or more (combinations of) `do`, `re` and `no`'s. This habit is a direct effect of the early days of writing macros. For TeX it does not matter how long a name is, as internally it becomes a pointer anyway, but memory consumption of editors, loading time of a format, string space and similar factors determined the way one codes in TeX for quite a while. Nowadays there are hardly any limits and the stress that ConTeXt puts on the TeX engine is even less than in MkII as we delegate many tasks to Lua. Memory comes cheap, editors can deal with large amount of data (keep in mind that the larger the file gets, the more lexing power can be needed), and screens are wide enough not to lose part of long names in the edges.

Another development has been that in LuaTeX we have lots of registers so that we no longer have to share temporary variables and such. The rewrite is a good moment to get rid of that restriction.

This all means that at some point it was decided to start using longer command names internally and permit `_` in names. As I was never a fan of using `@` for this, underscore made sense. We have been discussing the use of colons, which is also nice, but has the disadvantage that colons are also used in the source, for instance to create a sub-namespace. When we have replaced all old namespaces, colons might show up in command names, so another renaming roundup can happen.

One reason for mentioning this is that users get to see these names as part of error messages. An example of a name is:

```
\page_layouts_this_or_that
```

The first part of the name is the category of macros and in most cases is the same as the first part of the filename. The second part is a namespace. The rest of the name can differ but we're approaching some consistency in this.

In addition we have prefixed names, where prefixes are used as consistently as possible:

`t_` token register
`d_` dimension register
`s_` skip register
`u_` muskip register
`c_` counter register, constant or conditional
`m_` (temporary) macro
`p_` (temporary) parameter expansion (value of key)
`f_` fractions

This is not that different from other prefixing in ConTeXt apart from the fact that from now on those variables (registers) are no longer accessible in a regular run. We might decide on another scheme but renaming can easily be scripted. In the process some of the old prefixes are being removed. The main reason for changing to this naming scheme is that it is more convenient to grep for them.

In the process most traditional `\if`s get replaced by 'conditionals'. The same is true for `\chardef`s that store states; these become 'constants'.

23.5 Status

We always try to keep the user interface constant, so most functionality and control stays stable. However, now that most users use MkIV, commands that no longer make sense are removed. An interesting observation is that some users report that low-level macros or registers are no longer accessible. Fortunately that is no big deal as we point them to the official ways to deal with matters. It is also a good opportunity for users to clean up accumulated hackery.

The systematic (file by file) cleanup started in the second half of 2011 and as of January 2012 one third of the core (TeX) modules have to be cleaned up and the planning is to get most of that done as soon as possible. However, some modules will be rewritten (or replaced) and that takes more time. In any case we hope that rather soon most of the code is stable enough that we can start working on new mechanisms and features. Before that a cleanup of the Lua code is planned.

Although in many cases there are no fundamental changes in the user interface and functionality, I will wrap up some issues that are currently being dealt with. This is just a snapshot of what is happening currently and as a consequence it describes what users can run into due to newly introduced bugs.

The core modules of ConTeXt are loosely organized in groups. Over time there has been some reorganization and in MkIV some code has been moved into new categories. The alphabetical order does not reflect the loading order or dependency tree as categories are loaded intermixed. Therefore the order below is somewhat arbitrary and does not express importance. Each category has multiple files.

anch: anchoring and positioning

More than a decade ago we started experimenting with position tracking. The ability to store positional information and use that in a second pass permits for instance adding backgrounds. As this code interacts nicely with (runtime) MetaPost it has always been quite powerful and flexible on the one hand, but at the same time it was demanding in terms of runtime and resources. However, were it not for this feature, we would probably not be using TeX at all, as backgrounds and special relative positioning are needed in nearly all our projects.

In MkIV this mechanism had already been ported to a hybrid form, but recently much of the code has been overhauled and its MkII artifacts stripped. As a consequence the overhead in terms of memory probably has increased but the impact on runtime has been considerably reduced. It will probably take some time to become stable if only because the glue to MetaPost has changed. There are some new goodies, like backgrounds behind parshapes, something that probably no one uses and is always somewhat tricky but it was not too hard to support. Also, local background support has been improved which means that it's easier to get them in more column-based layouts, several table mechanisms, floats and such. This was always possible but is now more automatic and hopefully more intuitive.

attr: attributes

We use attributes (properties of nodes) a lot. The framework for this had been laid early in MkIV development, so not much has changed here. Of course the code gets cleaner and hopefully better as it is putting quite a load on the processing. Each new feature depending on attributes adds some extra overhead even if we make sure that mechanisms only kick in when they are used. This is due to the fact that attributes are linked lists and although unique lists are shared, they travel with each node. On the other hand, the cleanup (and de-

MkII-ing) of code leads to better performance so on the average no user will notice this.

back: backend code generation

This category wraps backend issues in an abstract way that is similar to the special drivers in MkII. So far we have only three backends: PDF, XML, and XHTML. Such code is always in a state of maintenance, if only because backends evolve.

bibl: bibliographies

For a while now, bibliographies have not been an add-on but part of the core. There are two variants: traditional BIBTEX support derived from a module by Taco Hoekwater but using MkIV features (the module hooks into core code), and a variant that delegates most work to LUA by creating an in-memory XML tree that gets manipulated. At some point I will extend the second variant. Going the XML route also connects better with developments such as Jean-Michel Hufflen's MLBIBTEX.

blob: typesetting in LUA

Currently we only ship a few helpers but eventually this will become a framework for typesetting raw text in LUA. This might be handy for some projects that we have where the only input is XML, but I'm not that sure if it will produce nice results and if the code will look better. On the other hand, there are some cases where in a regular TEX run some basic typesetting in LUA might make sense. Of course I also need an occasional pet project so this might qualify as one.

buff: buffers and verbatim

Traditionally buffers and verbatim have always been relatives as they share code. The code was among the first to be adapted to LUATEX. There is not that much to gain in adapting it further. Maybe I will provide more lexers for pretty-printing some day.

catc: catcodes

Catcodes are a rather TEX-specific feature and we have organized them in catcode regimes. The most important recent change has been that some of the characters with a special meaning in TEX (like ampersand, underscore, superscript, etc.) are no longer special except in cases that matter. This somewhat incompatible change surprisingly didn't lead to many problems. Some code that is specific for the MkII XML processor has been removed as we no longer assume it is in MkIV.

char: characters

This important category deals with characters and their properties. Already from the beginning of MkIV character properties have been (re)organized in Lua tables and therefore much code deals with it. The code is rather stable but occasionally the tables are updated as they depend on developments in Unicode. In order to share as much data as possible and prevent duplicates there are several inheritance mechanisms in place but their overhead is negligible.

chem: chemistry

The external module that deals with typesetting chemistry was transformed into a MkIV core module some time ago. Not much has changed in this department but some enhancements are pending.

cldf: CONTEXT LUA documents

These modules are mostly Lua code and are the interface into ConTeXt as well as providing ways to code complete documents in Lua. This is one of those categories that is visited every now and then to be adapted to improvements in other core code or in LuaTeX. This is one of my favourite categories as it exposes most of ConTeXt at the Lua end which permits writing solutions in Lua while still using the full power of ConTeXt. A dedicated manual is on its way.

colo: colors and transparencies

This is rather old code, and apart from some cleanup not much has been changed here. Some macros that were seldom used have been removed. One issue that is still pending is a better interface to MetaPost as it has different color models and we have adapted code at that end. This has a rather low priority because in practice it is no real problem.

cont: runtime code

These modules contain code that is loaded at runtime, such as filename remapping, patches, etc. It does not make much sense to improve these.

core: all kinds of core code

Housekeeping is the main target of these modules. There are still some typesetting-related components here but these will move to other categories. This code is cleaned up when there is a need for it. Think of managing files, document project structure, module loading, environments, multipass data, etc.

data: file and data management

This category hosts only LUA code and hasn't been touched for a while. Here we deal with locating files, caching, accessing remote data, resources, environments, and the like.

enco: encodings

Because (font) encodings are gone, there is only one file in this category and that one deals with weird (composed or otherwise special) symbols. It also provides a few traditional TEX macros that users expect to be present, for instance to put accents over characters.

file: files

There is some overlap between this category and core modules. Loading files is always somewhat special in TEX as there is the TEX directory structure to deal with. Sometimes you want to use files in the so-called tree, but other times you don't. This category provides some management code for (selective) loading of document files, modules and resources. Most of the code works with accompanying LUA code and has not been touched for years, apart from some weeding and low-level renaming. The project structure code has mostly been moved to LUA and this mechanism is now more restrictive in the sense that one cannot misuse products and components in unpredictable ways. This change permits better automatic loading of cross references in related documents.

font: fonts

Without proper font support a macro package is rather useless. Of course we do support the popular font formats but nowadays that's mostly delegated to LUA code. What remains at the TEX end is code that loads and triggers a combination of fonts efficiently. Of course in the process text and math each need to get the proper amount of attention.

There is no longer shared code between MKII and MKIV. Both already had rather different low-level solutions, but recently with MKIV we went a step further. Of course it made sense to kick out commands that were only used for PDFTEX TYPE1 and XETEX OPENTYPE support but more important was the decision to change the way design sizes are supported.

In CONTEXT we have basic font definition and loading code and that hasn't conceptually changed much over the years. In addition to that we have so-called bodyfont environments and these have been made a bit more powerful in recent MKIV. Then there are typefaces, which are abstract combinations

of fonts and defining them happens in typescripts. This layered approach is rather flexible, and was greatly needed when we had all those font encodings (to be used in all kinds of combinations within one document). In MkIV, however, we already had fewer typescripts as font encodings are gone (also for Type1 fonts). However, there remained a rather large blob of definition code dealing with Latin Modern; large because it comes in design sizes.

As we always fall back on Latin Modern, and because we don't preload fonts, there is some overhead involved in resolving design size related issues and definitions. But, it happens that this is the only font that ships with many files related to different design sizes. In practice no user will change the defaults. So, although the regular font mechanism still provides flexible ways to define font file combinations per bodyfont size, resolving to the right best matching size now happens automatically via a so-called Lua font goodie file which brings down the number of definitions considerably. The consequence is that ConTeXt starts up faster, not only in the case of Latin Modern being used, but also when other designs are in play. The main reason for this is that we don't have to parse those large typescripts anymore, as the presets were always part of the core set of typescripts. At the same time loading a specific predefined set has been automated and optimized. Of course on a run of 30 seconds this is not that noticeable, but it is on a 5 second run or when testing something in the editor that takes less than a second. It also makes a difference in automated workflows; for instance at Pragma ADE we run unattended typesetting flows that need to run as fast as possible. Also, in virtual machines using network shares, the fewer files consulted the better.

Because math support was already based on OpenType, where ConTeXt turns Type1 fonts into OpenType at runtime, nothing fundamental has changed here, apart from some speedups (at the cost of some extra memory). Where the overhead of math font switching in MkII is definitely a factor, in MkIV it is close to negligible, even if we mix regular, bold, and bidirectional math, which we have done for a while.

The low-level code has been simplified a bit further by making a better distinction between the larger sizes (a up to d) and smaller sizes (x and xx). These now operate independently of each other (i.e. one can now have a smaller relative x size of a larger one). This goes at the cost of more resources but it is worth the effort.

By splitting up the large basic font module into smaller ones, I hope that it can be maintained more easily although someone familiar with the older code will only recognize bits and pieces. This is partly due to the fact that font code is highly optimized.

grph: graphic (and widget) inclusion

Graphics inclusion is always work in progress as new formats have to be dealt with or users want additional conversions to be done. This code will be cleaned up later this year. The plug-in mechanisms will be extended (examples of existing plug-ins are automatic converters and barcode generation).

hand: special font handling

As we treat protrusion and hz as features of a font, there is not much left in this category apart from some fine-tuning. So, not much has happened here and eventually the left-overs in this category might be merged with the font modules.

java: JAVASCRIPT in PDF

This code already has been cleaned up a while ago, when moving to MkIV, but we occasionally need to check and patch due to issues with JavaScript engines in viewers.

lang: languages and labels

There is not much changed in this department, apart from additional labels. The way inheritance works in languages differs too much from other inheritance code so we keep what we have here. Label definitions have been moved to Lua tables from which labels at the TEX end are defined that can then be overloaded locally. Of course the basic interface has not changed as this is typically code that users will use in styles.

luat: housekeeping

This is mostly Lua code needed to get the basic components and libraries in place. While the `data` category implements the connection to the outside world, this category runs on top of that and feeds the TEX machinery. For instance conversion of MkVI files happens here. These files are seldom touched but might need an update some time (read: prune obsolete code).

lpdf: PDF backend

Here we implement all kinds of PDF backend features. Most are abstracted via the backend interface. So, for instance, colors are done with a high level command that goes via the backend interface to the `lpdf` code. In fact, there is more such code than in (for instance) the MkII special drivers, but readability comes at a price. This category is always work in progress as insights evolve

and users demand more.

lxml: XML and lpath

As this category is used by some power users we cannot change too much here, apart from speedups and extensions. It's also the bit of code we use frequently at PRAGMA ADE, and as we often have to deal with rather crappy XML I expect to move some more helpers into the code. The latest greatest trickery related to proper typesetting can be seen in the documents made by Thomas Schmitz. I wonder if I'd still have fun doing our projects if I hadn't, in an early stage of MKIV, written the XML parser and expression parser used for filtering.

math: mathematics

Math deserves its own category but compared to MKII there is much less code, thanks to UNICODE. Since we support TYPE1 as virtual OPENTYPE nothing special is needed there (and eventually there will be proper fonts anyway). When rewriting code I try to stay away from hacks, which is sometimes possible by using LUA but it comes with a slight speed penalty. Much of the UNICODE math-related font code is already rather old but occasionally we add new features. For instance, because OPENTYPE has no italic correction we provide an alternative (mostly automated) solution.

On the agenda is more structural math encoding (maybe like openmath) but tagging is already part of the code so we get a reasonable export. Not that someone is waiting for it, but it's there for those who want it. Most math-related character properties are part of the character database which gets extended on demand. Of course we keep MATHML up-to-date because we need it in a few projects.

We're not in a hurry here but this is something where Aditya and I have to redo some of the code that provides AMS-like math commands (but as we have them configurable some work is needed to keep compatibility). In the process it's interesting to run into probably never-used code, so we just remove those artifacts.

meta: metapost interfacing

This and the next category deal with MetaPost. This first category is quite old but already adapted to the new situation. Sometimes we add extra functionality but the last few years the situation has become rather stable with the exception of backgrounds, because these have been overhauled completely.

mlib: metapost library

Apart from some obscure macros that provide the interface between front- and backend this is mostly Lua code that controls the embedded MetaPost library. So, here we deal with extensions (color, shading, images, text, etc.) as well as runtime management because sometimes two runs are needed to get a graphic right. Some time ago, the MkII-like extension interface was dropped in favor of one more natural to the library and MetaPost 2. As this code is used on a daily basis it is quite well debugged and the performance is pretty good too.

mult: multi-lingual user interface

Even if most users use the English user interface, we keep the other ones around as they're part of the trademark. Commands, keys, constants, messages and the like are now managed with Lua tables. Also, some of the tricky remapping code has been stripped because the setup definitions files are dealt with. These are xml files that describe the user interface that get typeset and shipped with ConTeXt.

These files are being adapted. First of all the commandhandler code is defined here. As we use a new namespace model now, most of these namespaces are defined in the files where they are used. This is possible because they are more verbose so conflicts are less likely (also, some checking is done to prevent reuse). Originally the namespace prefixes were defined in this category but eventually all that code will be gone. This is a typical example where 15-year-old constraints are no longer an issue and better code can be used.

node: nodes

This is a somewhat strange category as all typeset material in TeX becomes nodes so this deals with everything. One reason for this category is that new functionality often starts here and is sometimes shared between several mechanisms. So, for the moment we keep this category. Think of special kerning, insert management, low-level referencing (layer between user code and backend code) and all kinds of rule and displacement features. Some of this functionality is described in previously published documents.

norm: normalize primitives

We used to initialize the primitives here (because LuaTeX starts out blank). But after moving that code this category only has one definition left and that one will go too. In MkII these files are still used (and actually generated by MkIV).

pack: wrapping content in packages

This is quite an important category as in ConTeXt lots of things get packed. The best example is `\framed` and this macro has been maximally optimized, which is not that trivial since much can be configured. The code has been adapted to work well with the new commandhandler code and in future versions it might use the commandhandler directly. This is however not that trivial because hooking a setup of a command into `\framed` can conflict with the two commands using keys for different matters.

Layers are also in this category and they probably will be further optimized. Reimplementing reusable objects is on the horizon, but for that we need a more abstract Lua interface, so that will come first. This has a low priority because it all works well. This category also hosts some helpers for the page builder but the builder itself has a separate category.

page: pages and output routines

Here we have an old category: output routines (trying to make a page), page building, page imposition and shipout, single and multi column handling, very special page construction, line numbering, and of course setting up pages and layouts. All this code is being redone stepwise and stripped of old hacks. This is a cumbersome process as these are core components where side effects are sometimes hard to trace because mechanisms (and user demands) can interfere. Expect some changes for the good here.

phys: physics

As we have a category for chemistry it made sense to have one for physics and here is where the unit module's code ended up. So, from now on units are integrated into the core. We took the opportunity to rewrite most of it from scratch, providing a bit more control.

prop: properties

The best-known property in TeX is a font and color is a close second. Both have their own category of files. In MkII additional properties like backend layers and special rendering of text were supported in this category but in MkIV properties as a generic feature are gone and replaced by more specific implementations in the `attr` namespace. We do issue a warning when any of the old methods are used.

regi: input encodings

We still support input encoding regimes but hardly any TEX code is involved now. Only when users demand more functionality does this code get extended. For instant, recently a user wanted a conversion function for going from UTF-8 to an encoding that another program wanted to see.

scrn: interactivity and widgets

All modules in this category have been overhauled. On the one hand we lifted some constraints, for instance the delayed initialization of fields no longer makes sense as we have a more dynamic variable resolver now (which is somewhat slower but still acceptable). On the other hand some nice but hard to maintain features have been simplified (not that anyone will notice as they were rather special). The reason for this is that vaguely documented PDF features tend to change over time which does not help portability. Of course there have also been some extensions, and it is actually less hassle (but still no fun) to deal with such messy backend related code in LUA.

scrp: script-specific tweaks

These are script-specific LUA files that help with getting better results for scripts like CJK. Occasionally I look at them but how they evolve depends on usage. I have some very experimental files that are not in the distribution.

sort: sorting

As sorting is delegated to LUA there is not much TEX code here. The LUA code occasionally gets improved if only because users have demands. For instance, sorting Korean was an interesting exercise, as was dealing with multiple languages in one index. Because sorting can happen on a combination of UNICODE, case, shape, components, etc. the sorting mechanism is one of the more complex subsystems.

spac: spacing

This important set of modules is responsible for vertical spacing, strut management, justification, grid snapping, and all else that relates to spacing and alignments. Already in an early stage vertical spacing was mostly delegated to LUA so there we're only talking of cleaning up now. Although ... I'm still not satisfied with the vertical spacing solution because it is somewhat demanding and an awkward mix of TEX and LUA which is mostly due to the fact that we cannot evaluate TEX code in LUA.

Horizontal spacing can be quite demanding when it comes down to configuration: think of a table with 1000 cells where each cell has to be set up (justification, tolerance, spacing, protrusion, etc.). Recently a more drastic optimization has been done which permits even more options but at the same time is much more efficient, although not in terms of memory.

Other code, for instance spread-related status information, special spacing characters, interline spacing and linewise typesetting all falls into this category and there is probably room for improvement there. It's good to mention that in the process of the current cleanup hardly any LUA code gets touched, so that's another effort.

strc: structure

Big things happened here but mostly at the TEX end as the support code in LUA was already in place. In this category we collect all code that gets or can get numbered, moves around and provides visual structure. So, here we find itemize, descriptions, notes, sectioning, marks, block moves, etc. This means that the code here interacts with nearly all other mechanisms.

Itemization now uses the new inheritance code instead of its own specific mechanism but that is not a fundamental change. More important is that code has been moved around, stripped, and slightly extended. For instance, we had introduced proper `\startitem` and `\stopitem` commands which are somewhat conflicting with `\item` where a next instance ends a previous one. The code is still not nice, partly due to the number of options. The code is a bit more efficient now but functionally the same.

The sectioning code is under reconstruction as is the code that builds lists. The intention is to have a better pluggable model and so far it looks promising. As similar models will be used elsewhere we need to converge to an acceptable compromise. One thing is clear: users no longer need to deal with arguments but variables and no longer with macros but with setups. Of course providing backward compatibility is a bit of a pain here.

The code that deals with descriptions, enumerations and notes was already done in a MKIV way, which means that they run on top of lists as storage and use the generic numbering mechanism. However, they had their own inheritance support code and moving to the generic code was a good reason to look at them again. So, now we have a new hierarchy: constructs, descriptions, enumerations and notations where notations are hooked into the (foot)note mechanisms.

These mechanisms share the rendering code but operate independently (which

was the main challenge). I did explore the possibility of combining the code with lists as there are some similarities but the usual rendering is too different as in the interface (think of enumerations with optional local titles, multiple notes that get broken over pages, etc.). However, as they are also stored in lists, users can treat them as such and reuse the information when needed (which for instance is just an alternative way to deal with end notes).

At some point math formula numbering (which runs on top of enumerations) might get its own construct base. Math will be revised when we consider the time to be ripe for it anyway.

The reference mechanism is largely untouched as it was already doing well, but better support has been added for automatic cross-document referencing. For instance it is now easier to process components that make up a product and still get the right numbering and cross referencing in such an instance.

Float numbering, placement and delaying can all differ per output routine (single column, multi-column, columnset, etc.). Some of the management has moved to Lua but most is just a job for TEX. The better some support mechanisms become, the less code we need here.

Registers will get the same treatment as lists: even more user control than is already possible. Being a simple module this is a relatively easy task, something for a hot summer day. General numbering is already fine as are block moves so they come last. The xml export and pdf tagging is also controlled from this category.

supp: support code

Support modules are similar to system ones (discussed later) but on a slightly more abstract level. There are not that many left now so these might as well become system modules at some time. The most important one is the one dealing with boxes. The biggest change there is that we use more private registers. I'm still not sure what to do with the visual debugger code. The math-related code might move to the math category.

symb: symbols

The symbol mechanisms organizes special characters in groups. With Unicode-related fonts becoming more complete we hardly need this mechanism. However, it is still the abstraction used in converters (for instance footnote symbols and interactive elements). The code has been cleaned up a bit but generally stays as is.

syst: tex system level code

Here you find all kinds of low-level helpers. Most date from early times but have been improved stepwise. We tend to remove obscure helpers (unless someone complains loudly) and add new ones every now and then. Even if we would strip down ConTeXt to a minimum size, these modules would still be there. Of course the bootstrap code is also in this category: think of allocators, predefined constants and such.

tabl: tables

The oldest table mechanism was a quite seriously patched version of TABLE and finally the decision has been made to strip, replace and clean up that bit. So, we have less code, but more features, such as colored columns and more.

The (in-stream) tabulate code is mostly unchanged but has been optimized (again) as it is often used. The multipass approach stayed but is somewhat more efficient now.

The natural table code was originally meant for xml processing but is quite popular among users. The functionality and code is frozen but benefits from optimizations in other areas. The reason for the freeze is that it is pretty complex multipass code and we don't want to break anything.

As an experiment, a variant of natural tables was made. Natural tables have a powerful inheritance model where rows and cells (first, last, ...) can be set up as a group but that is rather costly in terms of runtime. The new table variant treats each column, row and cell as an instance of `\framed` where cells can be grouped arbitrarily. And, because that is somewhat extreme, these tables are called x-tables. As much of the logic has been implemented in Lua and as these tables use buffers (for storing the main body) one could imagine that there is some penalty involved in going between TeX and Lua several times, as we have a two, three or four pass mechanism. However, this mechanism is surprisingly fast compared to natural tables. The reason for writing it was not only speed, but also the fact that in a project we had tables of 50 pages with lots of spans and such that simply didn't fit into TeX's memory any more, took ages to process, and could also confuse the float splitter.

Line tables ... well, I will look into them when needed. They are nice in a special way, as they can split vertically and horizontally, but they are seldom used. (This table mechanism was written for a project where large quantities of statistical data had to be presented.)

task: lua tasks

Currently this is mostly a place where we collect all kinds of tasks that are delegated to Lua, often hooked into callbacks. No user sees this code.

toks: token lists

This category has some helpers that are handy for tracing or manuals but no sane user will ever use them, I expect. However, at some point I will clean up this old MkIV mess. This code might end up in a module outside the core.

trac: tracing

A lot of tracing is possible in the Lua code, which can be controlled from the TEX end using generic enable and disable commands. At the macro level we do have some tracing but this will be replaced by a similar mechanism. This means that many `\tracewhatevertrue` directives will go away and be replaced. This is of course introducing some incompatibility but normally users don't use this in styles.

type: typescripts

We already mentioned that typescripts relate to fonts. Traditionally this is a layer on top of font definitions and we keep it this way. In this category there are also the definitions of typefaces: combinations of fonts. As we split the larger into smaller ones, there are many more files now. This has the added benefit that we use less memory as typescripts are loaded only once and stored permanently.

typo: typesetting and typography

This category is rather large in MkIV as we move all code into here that somehow deals with special typesetting. Here we find all kinds of interesting new code that uses Lua solutions (slower but more robust). Much has been discussed in articles as they are nice examples and often these are rather stable.

The most important new kid on the block is margin data, which has been moved into this category. The new mechanism is somewhat more powerful but the code is also quite complex and still experimental. The functionality is roughly the same as in MkII and older MkIV, but there is now more advanced inheritance, a clear separation between placement and rendering, slightly more robust stacking, local anchoring (new). It was a nice challenge but took a bit more time than other reimplementations due to all kinds of possible interference. Also, it's not always easy to simulate TEX grouping in a script language.

Even if much more code is involved, it looks like the new implementation is somewhat faster. I expect to clean up this code a couple of times.

On the agenda is not only further cleanup of all modules in this category, but also more advanced control over paragraph building. There is a parbuilder written in LUA on my machine for years already which we use for experiments and in the process a more LUATEX-ish (and efficient) way of dealing with protrusion has been explored. But for this to become effective, some of the LUATEX backend code has to be reorganized and Hartmut wants do that first. In fact, we can then backport the new approach to the built-in builder, which is not only faster but also more efficient in terms of memory usage.

unic: UNICODE vectors and helpers

As UNICODE support is now native all the MKII code (mostly vectors and converters) is gone. Only a few helpers remain and even these might go away. Consider this category obsolete and replaced by the `char` category.

util: utility functions

These are LUA files that are rather stable. Think of parsers, format generation, debugging, dimension helpers, etc. Like the data category, this one is loaded quite early.

Other TEX files

Currently there are the above categories which can be recognized by filename and prefix in macro names. But there are more files involved. For instance, user extensions can go into these categories as well but they need names starting with something like `xxxx-imp-` with `xxxx` being the category.

Then there are modules that can be recognized by their prefix: `m-` (basic module), `t-` (third party module), `x-` (XML-specific module), `u-` (user module), `p-` (private module). Some modules that Wolfgang and Aditya are working on might end up in the core distribution. In a similar fashion some seldom used core code might get moved to (auto-loaded) modules.

There are currently many modules that provide tracing for mechanisms (like font and math) and these need to be normalized into a consistent interface. Often such modules show up when we work on an aspect of CONTEXT or LUATEX and at that moment integration is not high on the agenda.

MetaPost files

A rather fundamental change in MetaPost is that it no longer has a format (mem file). Maybe at some point it will read `.gz` files, but all code is loaded at runtime.

For this reason I decided to split the files for MkII and MkIV as having version specific code in a common set no longer makes much sense. This means that already for a while we have `.mpii` and `.mpiv` files with the latter category being more efficient because we delegate some backend-related issues to ConTeXt directly. I might split up the files for MkIV a bit more so that selective loading is easier. This gives a slight performance boost when working over a network connection.

LUA files

There are some generic helper modules, with names starting with `l-`. Then there are the `mtx-*` scripts for all kinds of management tasks with the most important one being `mtx-context` for managing a TeX run.

Generic files

This leaves the bunch of generic files that provides OpenType support to packages other than ConTeXt. Much time went into moving ConTeXt-specific code out of the way and providing a better abstract interface. This means that new ConTeXt code (we provide more font magic) will be less likely to interfere and integration is easier. Of course there is a penalty for ConTeXt but it is bearable. And yes, providing generic code takes quite a lot of time so I sometimes wonder why I did it in the first place, but currently the maintenance burden is rather low. Khaled Hosny is responsible for bridging this code to LaTeX.

23.6 What next

Here ends this summary of the current state of ConTeXt. I expect to spend the rest of the year on further cleaning up. I'm close to halfway now. What I really like is that many users upgrade as soon as there is a new beta, and as in a rewrite typos creep in, I therefore often get a fast response.

Of course it helps a lot that Wolfgang Schuster, Aditya Mahajan, and Luigi Scarso know the code so well that patches show up on the list shortly after a problem gets reported. Also, for instance Thomas Schmitz uses the latest betas in academic book production, presentations, lecture notes and more, and so provides invaluable fast feedback. And of course Mojca Miklavec keeps all of it

(and us) in sync. Such a drastic cleanup could not be done without their help. So let's end this status report with ... a big thank you to all those (unnamed) patient users and contributors.

24 Just in time

24.1 Introduction

Reading occasional announcements about LuaJIT,[35] one starts wondering if just–in–time compilation can speed up LuaTEX. As a side track of the SwigLib project and after some discussion, Luigi Scarso decided to compile a version of LuaTEX that had the JIT compiler as the Lua engine. That's when our journey into JIT began.

We started with LINUX 32-bit as this is what Luigi used at that time. Some quick first tests indicated that the LuaJIT compiler made ConTEXt MkIV run faster but not that much. Because LuaJIT claims to be much faster than stock Lua, Luigi then played a bit with `ffi`, i.e. mixing C and Lua, especially data structures. There is indeed quite some speed to gain here; unfortunately, we would have to mess up the ConTEXt code base so much that one might wonder why Lua was used in the first place. I could confirm these observations in a Xubuntu virtual machine in VMWare running under 32-bit Windows 8. So, we decided to conduct some more experiments.

A next step was to create a 64-bit binary because the servers at Pragma ADE are KVM virtual machines running a 64-bit OpenSuse 12.1 and 12.2. It took a bit of effort to get a JIT version compiled because Luigi didn't want to mess up the regular codebase too much. This time we observed a speedup of about 40% on some runs so we decided to move on to Windows to see if we could observe a similar effect there. And indeed, when we adapted Akira Kakuto's Windows setup a bit we could compile a version for Windows using the native Microsoft compiler. On my laptop a similar speedup was observed, although by then we saw that in practice a 25% speedup was about what we could expect. A bonus is that making formats and identifying fonts is also faster.

So, in that stage, we could safely conclude that LuaTEX combined with LuaJIT made sense if you want a somewhat faster version. But where does the speedup come from? The easiest way to see if jitting has effect is to turn it on and off.

```
jit.on()
jit.off()
```

To our surprise ConTEXt runs are not much influenced by turning the jitter on or off.[36] This means that the improvement comes from other places:

[35] LuaJIT is written by Mike Pall and more information about it and the technology it uses is at `http://luajit.org`, a site also worth visiting for its clean design.

1. The virtual machine is a different one, and targets the platforms that it runs on. This means that regular bytecode also runs faster.
2. The garbage collector is the one from Lua 5.2, so that can make a difference. It looks like memory consumption is somewhat lower.
3. Some standard library functions are recognized and supported in a more efficient way. Think of `math.sin`.
4. Some built-in functions like `type` are probably dealt with in a more efficient way.

The third item is an important one. We don't use that many standard functions. For instance, if we need to go from characters to bytes and vice versa, we have to do that for utf so we use some dedicated functions or lpeg. If in ConTeXt we parse strings, we often use lpeg instead of string functions anyway. And if we still do use string functions, for instance when dealing with simple strings, it only happens a few times.

The more demanding ConTeXt code deals with node lists, which means frequent calls to core LuaTeX functions. Alas, jitting doesn't help much there unless we start messing with `ffi` which is not on the agenda.[37]

24.2 Benchmarks

Let's look at some of the benchmarks. The first one uses MetaPost and because we want to see if calculations are faster, we draw a path with a special pen so that some transformations have to be done in the code that generates the pdf output. We only show the MS Windows and 64-bit linux tests here. The 32-bit tests are consistent with those on MS Windows so we didn't add those timings here (also because in the meantime Luigi's machine broke down and he moved on to 64 bits).

```
\setupbodyfont[dejavu]

\starttext

\dontcomplain

\startluacode
    if jit then
```

[36] We also tweaked some of the fine-tuning parameters of LuaJIT but didn't notice any differences. In due time more tests will be done.

[37] If we want to improve these mechanisms it makes much more sense to make more helpers. However, profiling has shown us that the most demanding code is already quite optimized.

```
        jit.on()
        jit.off()
    end
\stopluacode

\startluacode
    statistics.starttiming()
\stopluacode

\dorecurse {10} {
    \dorecurse{1000} {
        \dontleavehmode
        \startMPcode
            for i=1,100 :
                draw
                    fullcircle scaled 10pt
                    withpen pencircle xscaled 2 yscaled 4 rotated 20 ;
            endfor ;
        \stopMPcode
        \enspace
    }
    \page
}

\startluacode
    statistics.stoptiming()
    context(statistics.elapsedtime())
\stopluacode

\stoptext
```

The following times are measured in seconds. They are averages of 5 runs. There is a significant speedup but jitting doesn't do much.

	traditional	JIT on	JIT off
Windows 8	26.0	20.6	20.8
Linux 64	34.2	14.9	14.1

Our second example uses multiple fonts in a paragraph and adds color as well. Although well optimized, font–related code involves node list parsing and a bit of calculation. Color again deals with node lists and the backend code involves calculations but not that many. The traditional run on LINUX is somewhat odd, but might have to do with the fact that the MetaPost library suffers from the

64 bits. It is at least an indication that optimizations make less sense if there is a different dominant weak spot. We have to look into this some time.

```
\setupbodyfont[dejavu]

\starttext

\dontcomplain

\startluacode
    if jit then
        jit.on()
        jit.off()
    end
\stopluacode

\startluacode
    statistics.starttiming()
\stopluacode

\dorecurse {1000} {
    {\bf \red   \input tufte } \blank
    {\it \green \input tufte } \blank
    {\tf \blue  \input tufte } \page
}

\startluacode
    statistics.stoptiming()
    context(statistics.elapsedtime())
\stopluacode

\stoptext
```

Again jitting has no real benefits here, but the overall gain in speed is quite nice. It could be that the garbage collector plays a role here.

	traditional	JIT on	JIT off
Windows 8	54.6	36.0	35.9
Linux 64	46.5	32.0	31.7

This benchmark writes quite a lot of data to the console, which can have impact on performance as TEX flushes on a per–character basis. When one runs TEX as a service this has less impact because in that case the output goes into the

void. There is a lot of file reading going on here, but normally the operating system will cache data, so after a first run this effect disappears.[38]

The third benchmark is one that we often use for testing regression in speed of the ConTeXt core code. It measures the overhead in the page builder without special tricks being used, like backgrounds. The document has some 1000 pages.

```
\setupbodyfont[dejavu]

\starttext

\dontcomplain

\startluacode
    if jit then
        jit.on()
        jit.off()
    end
\stopluacode

\startluacode
    statistics.starttiming()
\stopluacode

\dorecurse {1000} {
    test \page
}

\startluacode
    statistics.stoptiming()
    context(statistics.elapsedtime())
\stopluacode

\stoptext
```

These numbers are already quite okay for the normal version but the speedup of the LuaJIT version is consistent with the expectations we have by now.

	traditional	JIT on	JIT off

[38] On MS Windows it makes sense to use `console2` because due to some clever buffering tricks it has a much better performance than the default console.

Windows 8	4.5	3.6	3.6
Linux 64	4.8	3.9	4.0

The fourth benchmark uses some structuring, which involved LUA tables and housekeeping, an itemize, which involves numbering and conversions, and a table mechanism that uses more LUA than TEX.

```
\setupbodyfont[dejavu]

\starttext

\dontcomplain

\startluacode
    if jit then
        jit.on()
        jit.off()
    end
\stopluacode

\startluacode
    statistics.starttiming()
\stopluacode

\startbuffer
    \margintext{test} test test

    \startitemize[a]
        \startitem test \stopitem
        \startitem test \stopitem
        \startitem test \stopitem
        \startitem test \stopitem
    \stopitemize

    \startxtable
        \startxrow
            \startxcell test \stopxcell
            \startxcell test \stopxcell
            \startxcell test \stopxcell
        \stopxrow
        \startxrow
            \startxcell test \stopxcell
            \startxcell test \stopxcell
```

```
            \startxcell test \stopxcell
        \stopxrow
    \stopxtable
\stopbuffer

\dorecurse {25} {
    \startchapter[title=Test #1]
        \dorecurse {25} {
            \startsection[title=Test #1]
                \getbuffer
            \stopsection
        }
    \stopchapter
}

\page

\startluacode
    statistics.stoptiming()
    context(statistics.elapsedtime())
\stopluacode

\stoptext
```

Here it looks like JIT slows down the process, but of course we shouldn't take the last digit too seriously.

	traditional	JIT on	JIT off
Windows 8	20.9	16.8	16.5
Linux 64	20.4	16.0	16.1

Again, this example does a bit of logging, but not that much reading from file as buffers are kept in memory.

We should start wondering when JIT does kick in. This is what the fifth benchmark does.

```
\starttext

\startluacode

    if jit then
        jit.on()
```

```
        jit.off()
    end

    local t = os.clock()
    local a = 0
    for i=1,10*1000*1000 do
        a = a + math.sin(i)
    end
    context(os.clock()-t)

    context.par()

    local t = os.clock()
    local sin = math.sin
    local a = 0
    for i=1,10*1000*1000 do
        a = a + sin(i)
    end
    context(os.clock()-t)

\stopluacode

\stoptext
```

Here we see JIT having an effect! First of all the LuaJIT versions are now 4 times faster. Making the `sin` a `local` function (the numbers after /) does not make much of a difference because the math functions are optimized anyway.. See how we're still faster when JIT is disabled:

	traditional	JIT on	JIT off
Windows 8	1.97 / 1.54	0.46 / 0.45	0.73 / 0.61
Linux 64	1.62 / 1.27	0.41 / 0.42	0.67 / 0.52

Unfortunately this kind of calculation (in these amounts) doesn't happen that often but maybe some users can benefit.

24.3 Conclusions

So, does it make sense to complicate the LuaTEX build with LuaJIT? It does when speed matters, for instance when ConTEXt is run as a service. Some 25% gain in speed means less waiting time, better use of CPU cycles, less energy consumption, etc. On the other hand, computers are still becoming faster and

compared to those speed-ups the 25% is not that much. Also, as TEX deals with files, the advance of SSD disks and larger and faster memory helps too. Faster and larger CPU caches contributes too. On the other hand, multiple cores don't help that much on a system that only runs TEX. Interesting is that multi-core architectures tend to run at slower speeds than single cores where more heat can be dissipated and in that respect servers mostly running TEX are better off with fewer cores that can run at higher frequencies. But anyhow, 25% is still better than nothing and it makes my old laptop feel faster. It prolongs the lifetime of machines!

Now, say that we cannot speed up TEX itself that much, but that there is still something to gain at the LUA end — what can we reasonably expect? First of all we need to take into account that only part of the runtime is due to LUA. Say that this is 25% for a document of average complexity.

$$\text{runtime}_{\text{tex}} + \text{runtime}_{\text{lua}} = 100$$

We can consider the time needed by TEX to be constant; so if that is 75% of the total time (say 100 seconds) to begin with, we have:

$$75 + \text{runtime}_{\text{lua}} = 100$$

It will be clear that if we bring down the runtime to 80% (80 seconds) of the original we end up with:

$$75 + \text{runtime}_{\text{lua}} = 80$$

And the 25 seconds spent in LUA went down to 5, meaning that LUA processing got 5 times faster! It is also clear that getting much more out of LUA becomes hard. Of course we can squeeze more out of it, but TEX still needs its time. It is hard to measure how much time is actually spent in LUA. We do keep track of some times but it is not that accurate. These experiments and the gain in speed indicate that we probably spend more time in LUA than we first guessed. If you look in the CONTEXT source it's not that hard to imagine that indeed we might well spend 50% or more of our time in LUA and/or in transferring control between TEX and LUA. So, in the end there still might be something to gain.

Let's take benchmark 4 as an example. At some point we measured for a regular LUATEX 0.74 run 27.0 seconds and for a LUAJITTEX run 23.3 seconds. If we assume that the LUAJIT virtual machine is twice as fast as the normal one, some juggling with numbers makes us conclude that TEX takes some 19.6 seconds of this. An interesting border case is `\directlua`: we sometimes pass quite a lot of data and that gets tokenized first (a TEX activity) and the resulting token list is converted into a string (also a TEX activity) and then converted to

bytecode (a Lua task) and when okay executed by Lua. The time involved in conversion to byte code is probably the same for stock Lua and LuaJIT.

In the LuaTEX case, 30% of the runtime for benchmark 4 is on Lua's tab, and in LuaJITTEX it's 15%. We can try to bring down the Lua part even more, but it makes more sense to gain something at the TEX end. There macro expansion can be improved (read: ConTEXt core code) but that is already rather optimized.

Just for the sake of completeness Luigi compiled a stock LuaTEX binary for 64-bit linux with the `-o3` option (which forces more inlining of functions as well as a different switch mechanism). We did a few tests and this is the result:

	LuaTEX 0.74 -o2	LuaTEX 0.74 - o3
benchmark-1	15.5	15.0
benchmark-2	35.8	34.0
benchmark-3	4.0	3.9
benchmark-4	16.0	15.8

This time we used `--batch` and `--silent` to eliminate terminal output. So, if you really want to squeeze out the maximum performance you need to compile with `-o3`, use LuaJITTEX (with the faster virtual machine) but disable jit (disabled by default anyway).

We have no reason to abandon stock Lua. Also, because during these experiments we were still using Lua 5.1 we started wondering what the move to 5.2 would bring. Such a move forward also means that ConTEXt MkIV will not depend on specific LuaJIT features, although it is aware of it (this is needed because we store bytecodes). But we will definitely explore the possibilities and see where we can benefit. In that respect there will be a way to enable and disable jitting. So, users have the choice to use either stock LuaTEX or the jit–aware version but we default to the regular binary.

As we use stock Lua as benchmark, we will use the `bit32` library, while LuaJIT has its own bit library. Some functions can be aliased so that is no big deal. In ConTEXt we use wrappers anyway. More problematic is that we want to move on to Lua 5.2 and not all 5.2 features are supported (yet) in LuaJIT. So, if LuaJIT is mandatory in a workflow, then users had better make sure that the Lua code is compatible. We don't expect too many problems in ConTEXt MkIV.

24.4 About speed

It is worth mentioning that the Lua version in LuaTEX has a patch for converting

floats into strings. Instead of some `INF#` result we just return zero, simply because TEX is integer–based and intercepting incredibly small numbers is too cumbersome. We had to apply the same patch in the JIT version.

The benchmarks only indicate a trend. In a real document much more happens than in the above tests. So what are measurements worth? Say that we compile the TEXbook. This grandparent of all documents coded in TEX is rather plainly coded (using of course plain TEX) and compiles pretty fast. Processing does not suffer from complex expansions, there is no color, hardly any text manipulation, it's all 8 bit, the pagebuilder is straightforward as is all spacing. Although on my old machine I can get CONTEXT to run at over 200 pages per second, this quickly drops to 10% of that speed when we add some color, backgrounds, headers and footers, font switches, etc.

So, running documents like the TEXbook for comparing the speed of, say, PDFTEX, XETEX, LUATEX and now LUAJITTEX makes no sense. The first one is still eight bit, the rest are UNICODE. Also, the TEXbook uses traditional fonts with traditional features so effectively that it doesn't rely on anything that the new engines provide, not even ε-TEX extensions. On the other hand, a recent document uses advanced fonts, properties like color and/or transparencies, hyperlinks, backgrounds, complex cover pages or chapter openings, embeds graphics, etc. Such a document might not even process in PDFTEX or XETEX, and if it does, it's still comparing different technologies: eight bit input and fast fonts in PDFTEX, frozen UNICODE and wide font support in XETEX, instead of additional trickery and control, written in LUA. So, when we investigate speed, we need to take into account what (font and input) technologies are used as well as what complicating layout and rendering features play a role. In practice speed only matters in an edit-view cycle and services where users wait for some result.

It's rather hard to find a recent document that can be used to compare these engines. The best we could come up with was the rendering of the user interface documentation.

```
texexec  --engine=pdftex     --global  x-set-12.mkii  5.9 seconds
texexec  --engine=xetex      --global  x-set-12.mkii  6.2 seconds
context  --engine=luatex     --global  x-set-12.mkiv  6.2 seconds
context  --engine=luajittex  --global  x-set-12.mkiv  4.6 seconds
```

Keep in mind that `texexec` is a RUBY script and uses `kpsewhich` while `context` uses LUA and its own (TDS–compatible) file manager. But still, it is interesting to see that there is not that much difference if we keep JIT out of the picture. This is because in MKIV we have somewhat more clever XML processing, although earlier measurements have demonstrated that in this case not that much speedup can be assigned to that.

And so recent versions of MkIV already keep up rather well with the older eight bit world. We do way more in MkIV and the interfacing macros are nicer but potentially somewhat slower. Some mechanisms might be more efficient because of using Lua, but some actually have more overhead because we keep track of more data. Font feature processing is done in Lua, but somehow can keep up with the libraries used in XeTeX, or at least is not that significant a difference, although I can think of more demanding tasks. Of course in LuaTeX we can go beyond what libraries provide.

No matter what one takes into account, performance is not that much worse in LuaTeX, and if we enable jit and so remove some of the traditional Lua virtual machine overhead, we're even better off. Of course we need to add a disclaimer here: don't force us to prove that the relative speed ratios are the same for all cases. In fact, it being so hard to measure and compare, performance can be considered to be something taken for granted as there is not that much we can do about getting nicer numbers, apart from maybe parallelizing which brings other complexities into the picture. On our servers, a few other virtual machines running TeX services kicking in at the same time, using cpu cycles, network bandwidth (as all data lives someplace else) and asking for disk access have much more impact than the 25% we gain. Of course if all processes run faster then we've gained something.

For what it's worth: processing this text takes some 2.3 seconds on my laptop for regular LuaTeX and 1.8 seconds with LuajitTeX, including the extra overhead of restarting. As this is a rather average example it fits earlier measurements.

Processing a font manual (work in progress) takes LuajitTeX 15 seconds for 112 pages compared to 18.4 seconds for LuaTeX. The not yet finished manual loads 20 different fonts (each with multiple instances), uses colors, has some MetaPost graphics and does some font juggling. The gain in speed sounds familiar.

24.5 The future

At the 2012 Lua conference Roberto Ierusalimschy mentioned that the virtual machine of LuaJIT is about twice as fast due to it being partly done in assembler while the regular machinery is written in standard C and keeps portability in mind.

He also presented some plans for future versions of Lua. There will be some lightweight helpers for utf. Our experiences so far are that only a handful of functions are actually needed: byte to character conversions and vice versa,

iterators for UTF characters and UTF values and maybe a simple substring function is probably enough. Currently LuaTEX has some extra string iterators and it will provide the converters as well.

There is a good chance that LPEG will become a standard library (which it already is in LuaTEX), which is also nice. It's interesting that, especially on longer sequences, LPEG can beat the string matchers and replacers, although when in a substitution no match and therefore no replacements happen, the regular gsub wins. We're talking small numbers here, in daily usage LPEG is about as efficient as you can wish. In ConTEXt we have a `lpeg.UR` and `lpeg.US` and it would be nice to have these as native UTF related methods, but I must admit that I seldom need them.

This and other extensions coming to the language also have some impact on a JIT version: the current LuaJIT is already not entirely compatible with Lua 5.2 so you need to keep that into account if you want to use this version of LuaTEX. So, unless LuaJIT follows the mainstream development, as ConTEXt MkIV user you should not depend on it. But at the moment it's nice to have this choice.

The yet experimental code will end up in the main LuaTEX repository in time before the TEX Live 2013 code freeze. In order to make it easier to run both versions alongside, we have added the Lua 5.2 built-in library `bit32` to LuaJITTEX. We found out that it's too much trouble to add that library to Lua 5.1 but LuaTEX has moved on to 5.2 anyway.

24.6 Running

So, as we will definitely stick to stock Lua, one might wonder if it makes sense to officially support jitting in ConTEXt. First of all, LuaTEX is not influenced that much by the low level changes in the API between 5.1 and 5.2. Also LuaJIT does support the most important new 5.2 features, so at the moment we're mostly okay. We expect that eventually LuaJIT will catch up but if not, we are not in big trouble: the performance of stock Lua is quite okay and above all, it's portable![39] For the moment you can consider LuaJITTEX to be an experiment and research tool, but we will do our best to keep it production ready.

So how do we choose between the two engines? After some experimenting with alternative startup scenarios and dedicated caches, the following solution was reached:

[39] Stability and portability are important properties of TEX engines, which is yet another reason for using Lua. For those doing number crunching in a document, JIT can come in handy.

```
context --engine=luajittex ...
```

The usual preamble line also works:

```
% engine=luajittex
```

As the main infrastructure uses the `luatex` and related binaries, this will result in a relaunch: the `context` script will be restarted using `luajittex`. This is a simple solution and the overhead is rather minimal, especially compared to the somewhat faster run. Alternatively you can copy `luajittex` over `luatex` but that is more drastic. Keep in mind that `luatex` is the benchmark for development of ConTeXt, so the JIT aware version might fall behind sometimes.

Yet another approach is adapting the configuration file, or better, provide (or adapt) your own `texmfcnf.lua` in for instance `texmf-local/web2c` path:

```
return {
  type    = "configuration",
  version = "1.2.3",
  date    = "2012-12-12",
  time    = "12:12:12",
  comment = "Local overloads",
  author  = "Hans Hagen, PRAGMA-ADE, Hasselt NL",
  content = {
    directives = {
      ["system.engine"] = "luajittex",
    },
  },
}
```

This has the same effect as always providing `--engine=luajittex` but only makes sense in well controlled situations as you might easily forget that it's the default. Of course one could have that file and just comment out the directive unless in test mode.

Because the bytecode of LuaJIT differs from the one used by Lua itself we have a dedicated format as well as dedicated bytecode compiled resources (for instance `tmb` instead of `tmc`). For most users this is not something they should bother about as it happens automatically.

Based on experiments, by default we have disabled JIT so we only benefit from the faster virtual machine. Future versions of ConTeXt might provide some control over that but first we want to conduct more experiments.

24.7 Addendum

These developments and experiments took place in November and December 2012. At the time of this writing we also made the move to Lua 5.2 in stock LuaTeX; the first version to provide this was 0.74. Here are some measurements on Taco Hoekwater's 64-bit linux machine:

	LuaTeX 0.70	LuaTeX 0.74	
benchmark-1	23.67	19.57	faster
benchmark-2	65.41	62.88	faster
benchmark-3	4.88	4.67	faster
benchmark-4	23.09	22.71	faster
benchmark-5	2.56/2.06	2.66/2.29	slower

There is a good chance that this is due to improvements of the garbage collector, virtual machine and string handling. It also looks like memory consumption is a bit less. Some speed optimizations in reading files have been removed (at least for now) and some patches to the `format` function (in the `string` namespace) that dealt with (for TeX) unfortunate number conversions have not been ported. The code base is somewhat cleaner and we expect to be able to split up the binary in a core program plus some libraries that are loaded on demand.[40] In general, we don't expect too many issues in the transition to Lua 5.2, and ConTeXt is already adapted to support LuaTeX with 5.2 as well as LuajitTeX with an older version.

Running the same tests on a 32-bit MS Windows machine gives this:

	LuaTeX 0.70	LuaTeX 0.74	
benchmark-1	26.4	25.5	faster
benchmark-2	64.2	63.6	faster
benchmark-3	7.1	6.9	faster
benchmark-4	28.3	27.0	faster
benchmark-5	1.95/1.50	1.84/1.48	faster

The gain is less impressive but the machine is rather old and we can benefit less from modern cpu properties (cache, memory bandwidth, etc.). I tend to conclude that there is no significant improvement here but it also doesn't get worse. However we need to keep in mind that file io is less optimal in 0.74 so this might play a role. As usual, runtime is negatively influenced by the relatively slow speed of displaying messages on the console (even when we use

[40] Of course this poses some constraints on stability as components get decoupled, but this is one of the issues that we hope to deal with properly in the library project.

console2).

A few days before the end of 2012, Akira Kakuto compiled native MS Windows binaries for both engines. This time I decided to run a comparison inside the SciTE editor, that has very fast console output.[41]

	LuaTeX 0.74 (5.2)	LuaJITTeX 0.72 (5.1)	
benchmark-1	25.4	25.4	similar
benchmark-2	54.7	36.3	faster
benchmark-3	4.3	3.6	faster
benchmark-4	20.0	16.3	faster
benchmark-5	1.93/1.48	0.74/0.61	faster

Only the MetaPost library and conversion benchmark didn't show a speedup. The regular TeX tests 1–3 gain some 15–35%. Enabling jit (off by default) slowed down processing. For the sake of completeness I also timed LuaJITTeX on the console, so here you see the improvement of both engines.

	LuaTeX 0.70	LuaTeX 0.74	LuaJITTeX 0.72
benchmark-1	26.4	25.5	25.9
benchmark-2	64.2	63.6	45.5
benchmark-3	7.1	6.9	6.0
benchmark-4	28.3	27.0	23.3
benchmark-5	1.95/1.50	1.84/1.48	0.73/0.60

In this text, the term jit has come up a lot but you might rightfully wonder if the observations here relate to jit at all. For the moment I tend to conclude that the implementation of the virtual machine and garbage collection have more impact than the actual just–in–time compilation. More exploration of jit is needed to see if we can really benefit from that. Of course the fact that we use a bit less memory is also nice. In case you wonder why I bother about speed at all: we happen to run LuaTeX mostly as a (remote) service and generating a bunch of (related) documents takes a bit of time. Bringing the waiting down from 15 to 10 seconds might not sound impressive but it makes a difference when it is someone's job to generate these sets.

In summary: just before we entered 2013, we saw two rather fundamental updates of LuaTeX show up: an improved traditional one with Lua 5.2 as well as the somewhat faster LuaJITTeX with a mixture between 5.1 and 5.2. And in 2013 we will of course try to make them both even more attractive.

[41] Most of my personal TeX runs are from within SciTE, while most runs on the servers are in batch mode, so normally the overhead of the console is acceptable or even neglectable.

The team

The LuaTEX project started in 2005 as a follow up on some experiments. The core team consists of Taco Hoekwater, Hartmut Henkel and Hans Hagen, here pictured at work by Duane Bibby. The machine they work on is inspired by the Paige Typesetter (http://www.twainquotes.com/paige.html).