Zpravodaj Československého sdružení uživatelů TeXu

Hans Hagen
MkIV Hybrid Technology

Zpravodaj Československého sdružení uživatelů TeXu, Vol. 21 (2011), No. 2-4, 182–300

Persistent URL: http://dml.cz/dmlcz/150189

Terms of use:

© Československé sdružení uživatelů TeXu, 2011

Institute of Mathematics of the Czech Academy of Sciences provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.

This document has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* http://dml.cz

MkIV Hybrid Technology
Hybridní technologie MkIV

Hans Hagen

Abstract: The paper presents development, new features and tools of LuaTEX and ConTEXtMkIV.

Key words: LuaTEX, ConTEXtMkIV, Mark II, Mark IV.

Abstrakt: Příspěvek představuje rozvoj, nové vlastnosti a nástroje LuaTEXu a formátu ConTEXt Mark IV.

Klíčová slova: LuaTEX, ConTEXtMkIV, Mark II, Mark IV.

References

[1] LuaTEXhome page. Available at URL: `http://www.luatex.org/`
[2] The Programming Language Lua. Home page.
Available at URL: `http://www.lua.org/`

pragma (at) wxs (dot) nl
PRAGMA ADE, Ridderstraat 27
8061GH Hasselt, The Netherlands

doi: 10.5300/2011-2-4/182

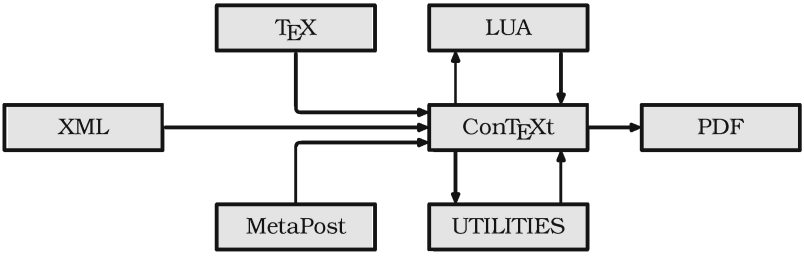
Introduction

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).

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

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

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]

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.

Hans Hagen, Hasselt NL,
September 2009 and beyond

`http://www.luatex.org`

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

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 that consist of such commands directly if they have a `cld` suffix or when you provide the flag `--forcecld`.[3]

[3] Similar methods exist for processing xml files.

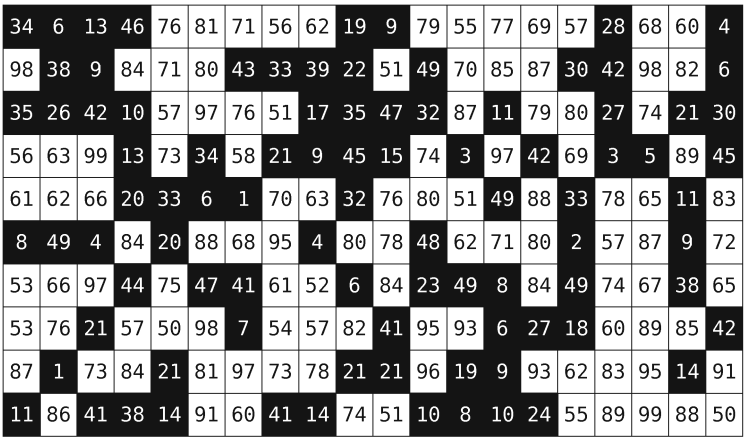


Figure 1.1 The result of the shown Lua code.

```
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.

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
```

[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.

The filter variant is like this:

```
\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).

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

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 distribution but it has been rewritten in Lua.

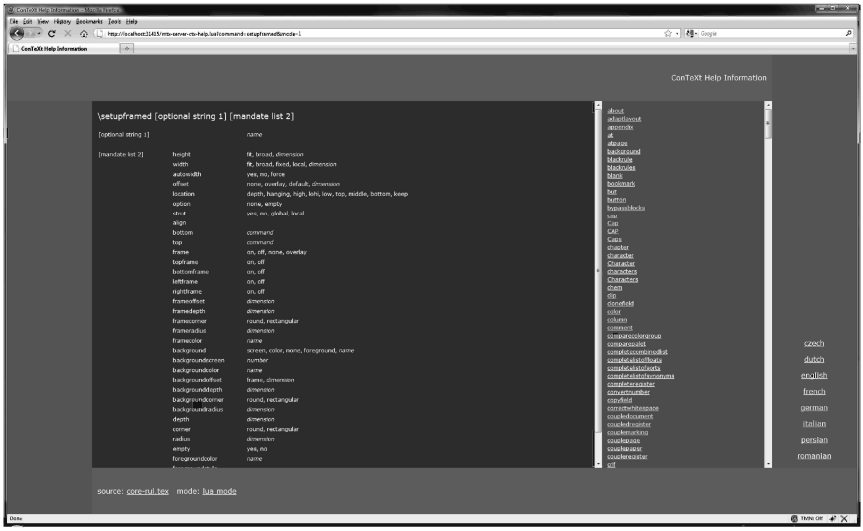


Figure 1.3 An example of a help screen for a command.

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.

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 passed 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)

first order

second order

second order (Jiim-stacking)

minimal stretching

maximal stretching (level 3)

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 ships 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
```

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:

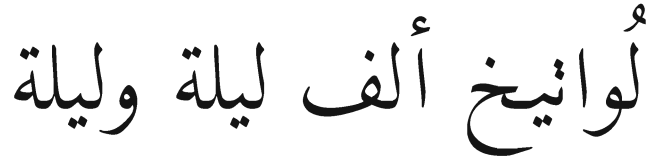
featureset	definitions
default	analyze=true anum=true calt=true ccmp=true curs=true fina=true init=true js16=true kern=true language=dflt mark=true medi=true mkmk=true mode=node number=32 rlig=true salt=true script=arab ss01=true ss03=true ss07=true ss10=true ss12=true ss15=true ss16=true ss19=true ss24=true ss25=true ss26=true ss27=true ss31=true ss34=true ss35=true ss36=true ss37=true ss38=true ss41=true ss42=true ss43=true
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=35 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
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=36 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
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=31 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

```
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=34  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
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=33
                    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
```

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.



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 ألف ليلة وليلة \par
  \setfontcolorscheme  [1]لُواتيخ ألف 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.get("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.get("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 close 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.

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

```
\definedfont[husayni*husayni-default at 24pt]
\expanded{\setuplocalinterlinespace[line=\the\dimexpr2\lineheight]} % todo:
```

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

```
factor ivm grid
\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 فالكلم 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 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.

Todo: show normal, reverse and random.

Todo: bind setting to paragraph.

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.

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 will 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 typesetting.
- 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 can we 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:

$$e = \quad e$$

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 that 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 languags 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 languags 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 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.

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 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

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

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.

5 Deeply nested notes

5.1 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 `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.

5.2 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 gets 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 vertical 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.

5.3 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)
```

5.4 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.

5.5 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.

6 Backend code

6.1 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.

6.2 Structure

A pdf file is tree of indirect objects. Each object has a number and the file contains a table (or more 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).
string	`(...)`	A sequence of characters in pdfdoc encoding
unicode	`<...>`	A sequence of characters in utf16 encoding

[8] At this moment we only support the native pdf backend but future versions might support xml (html) output as well.

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.stream`	indexed table of operators
`lpdf.dictionary`	hash wit 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.true`	
`lpdf.false`	
`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)
```

6.3 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 passed 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 that 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 a bit of tracing built in and we will add some more 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.

6.4 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.

6.5 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 resources of all kind (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.

6.6 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 codes 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 compleet 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 a suggest that you stay away from this metadata mess as far as possible.

6.7 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.cleaned(str)    returns escaped 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.

7 Callbacks

7.1 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.

7.2 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.

`buildpage_filter`	`enabled`	vertical spacing etc (mvl)
`char_exists`	`undefined`	
`define_font`	`enabled`	definition of fonts (tfmtable preparation)
`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`	
`hpack_filter`	`enabled`	all kind of horizontal manipulations
`hyphenate`	`disabled`	normal hyphenation routine, called elsewhere
`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_output_buffer	disabled	actions performed when writing data
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	
start_page_number	enabled	actions performed at the beginning of a shipout
start_run	enabled	actions performed at the beginning of a run
stop_page_number	enabled	actions performed at the end of a shipout
stop_run	enabled	actions performed at the end of a run
token_filter	undefined	
vpack_filter	enabled	vertical spacing etc

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 other ConTEXt behaviour 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 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]

[9] This interface might change a bit in future versions of ConTEXt. Therefore we will not discuss

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

7.3 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.

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

category	function
before	unset
normalizers	fonts.collections.process
	fonts.checkers.missing
characters	typesetters.directions.handler
	typesetters.cases.handler
	typesetters.breakpoints.handler
	scripts.preprocess
words	builders.kernel.hyphenation
	languages.words.check
fonts	builders.paragraphs.solutions.splitters.split
	nodes.handlers.characters
	nodes.injections.handler
	nodes.handlers.protectglyphs
	builders.kernel.ligaturing
	builders.kernel.kerning
	nodes.handlers.stripping

the few more optional arguments that are possible.

	fonts.goodies.colorschemes.coloring
lists	typesetters.spacings.handler typesetters.kerns.handler typesetters.digits.handler
after	unset

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	nodes.handlers.graphicvadjust
after	unset

Shipouts concern:

category	**function**
before	unset
normalizers	nodes.handlers.cleanuppage nodes.references.handler nodes.destinations.handler nodes.rules.handler nodes.shifts.handler structures.tags.handler nodes.handlers.accessibility nodes.handlers.backgrounds
finishers	attributes.colors.handler attributes.transparencies.handler attributes.colorintents.handler attributes.effects.handler attributes.viewerlayers.handler
after	unset

There are not that many mvlbuilder task currently:

category	**function**

before	unset
normalizers	streams.collect nodes.handlers.migrate builders.vspacing.pagehandler
after	unset

The vboxbuilder perform similar tasks:

category	**function**
before	unset
normalizers	builders.vspacing.vboxhandler
after	unset

Finally, we have tasks related to the math list:

category	**function**
before	unset
normalizers	noads.handlers.relocate noads.handlers.resize noads.handlers.respace noads.handlers.check noads.handlers.tags
builders	builders.kernel.mlist_to_hlist
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]

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

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

8 Bibliographies

8.1 Introduction

Already early in the history of ConTEXt Taco Hoekwater wrote a module that dealt with bibTEX databases in a ConTEXt like way. Personally I never had to use a bibliography so I'm far from aan expert in this area. However, going from some text database format to something typeset is generic enough for me to be involved.

The involvement started when MkIV showed up. Because quite some core mechanisms have been reimplemented, also some that the module used, a dedicated MkIV variant had to be made. This was not that hard to do as it mostly meant stripping code and replacing the specific reference mechanism by one using lists. That way we got a few bonus features but in general we can say that the module is downward compatible.

Already a while ago Taco and I discussed supporting bibliographies that use xml as format and although we have not settled on some standard it makes sense to explore the possibilities. The advantage of using xml is that we can use the built in subsystem for filtering and manipulating entries.

This chapter is dedicated to Thomas Schmitz who not only use bibTEX but also has used MkIV xml right from the start and provides valuable feedback on both subsystems.

Keep in mind that eventually we will provide a high level interface so that users won't notice much of a difference unless they want to go beyond what they use now.

8.2 Sessions

As usual in ConTEXt, we organize the featureset in such a way that we can group them and use several such sets in one documents without interference. It all starts by defining a session:

```
\definebibtexsession [somebibtex]
```

Next we register a couple of databases (from the beebe collection on TEXlive:

```
\registerbibtexfile [somebibtex] [tugboat.bib]
\registerbibtexfile [somebibtex] [komoedie.bib]
```

The files are loaded immediately and you can check this by looking at the log

where we get a report like:

```
mkiv lua stats : bibtex load time - 0.125 seconds (1524045 bytes, 2745 definitions, 7 shortcuts)
```

In a bibtex database there can be symbols (shortcuts to strings). Although these are expanded, it has no consequence for memory usage as Lua hashes its strings.

When we're done registering we need to prepare the session:[11]

```
\preparebibtexsession [somebibtex] % [convert]
```

Say that we want to typeset a table with the publications of where we only list the the author and title. As we use the xml interface we do so using setups:

```
\startxmlsetups bibtex:one
  \starttabulate[|Bl|p|]
  \NC tag   \NC \xmlatt{#1}{tag} \NC\NR
  \NC author\NC \xmlfilter{#1}{/field[@name='author']/context()} \NC\NR
  \NC title \NC \xmlfilter{#1}{/field[@name='title' ]/context()} \NC\NR
  \stoptabulate
\stopxmlsetups
```

We will now apply this setup to the loaded tree:

```
\startxmlsetups bibtex:bibtex
    \xmlfilter{#1}{
        /entry[@category='article']
        /field[@name='author' and (find(text(),'Hans Hagen')
            or find(text(),'Taco Hoekwater'))]
        /../command(bibtex:one)
    }
\stopxmlsetups
```

We now apply this setup to the session:

```
\applytobibtexsession[somebibtex][bibtex]
```

tag hoekwater:tb19-3-256
author Taco Hoekwater
title Generating Type 1 fonts from MF sources

[11] The convert option will be discussed later.

tag hagen:tb25-1-108
author Hans Hagen
title The TEX Live 2004 collection

tag hagen:tb19-3-304
author Hans Hagen
title The Calculator Demo, Integrating TEX, MP, JavaScript and PDF

tag hagen:tb19-3-311
author Hans Hagen
title Visual Debugging in TEX, Part 1: The Story

tag hagen:tb23-1-49
author Hans Hagen
title ConTEXt, XML and TEX: State of the art?

tag hagen:tb26-2-152
author Hans Hagen
title LuaTEX: Howling to the moon

tag hoekwater:tb25-1-105
author Taco Hoekwater
title MetaPost developments

tag hagen:tb25-1-48
author Hans Hagen
title The state of ConTEXt

tag hagen:tb22-3-136
author Hans Hagen
title Using TEX for high end typesetting

tag hagen:tb22-3-118
author Hans Hagen
title Where will the odyssey bring us?

tag hagen:tb22-1-58
author Hans Hagen
title The status quo of the nts project

tag berdnikov:tb21-2-129
author Alexander Berdnikov and Hans Hagen and Taco Hoekwater and Bogusław Jackowski
title Even more MetaFun with MP: A request for permission

tag hagen:tb19-3-317
author Hans Hagen
title Visual Debugging in TEX, Part 2: The Macros

tag hagen:tb22-3-160
author Hans Hagen
title Using TEX to enhance your presentations

If this is the first time you see MkIV's xmlin action you might be confused by what happens here. When we apply the `bibtex` setup (the second argument), we expand a predefined setup that looks as follows:

```
\startxmlsetups bibtex
    \xmlregistereddocumentsetups{#1}{}
    \xmlsetsetup{#1}{bibtex|entry|field}{bibtex:*}
    \xmlmain{#1}
\stopxmlsetups
```

Here `#1` represents the root node of current database. Three elements are mapped to their own name, prefixed by `bibtex:`. In the previous examples we defined the `bibtex:bibtex` one, which will be applied to the root.

Here are a few more predefined setups:

```
\startxmlsetups bibtex:format
    \par
    \edef\currentbibxmlnode{#1}
    \xmlcommand{#1}{.}{bibtex:\currentbibtexformat:\xmlatt{#1}{category}}
    \par
\stopxmlsetups

\startxmlsetups bibtex:list
    \xmlfilter{#1}{/bibtex/entry/command(bibtex:format)}
\stopxmlsetups

\startxmlsetups bibtex:bibtex
    \xmlfilter{#1}{/entry/command(bibtex:format)}
\stopxmlsetups
```

The first one apply a setup to the current node (indicated by the period).

```
bibtex:apa:article
```

Such setups are defined elsewhere and you can imagine that they look more

complex than what we've seen sofar. But you seldom have to deal with that.

The second and third setups applie the format to an entry. However, there is a subtle difference. The second one is called as follows:

```
\applytobibtexsession[somebibtex][bibtex:list]
```

As `bibtex:list` is a stand-alone setup, it will get the document root passed, and therefore we need to explicitly add that root, although the following two calls give the same results (watch the forward slashes):

```
\xmlfilter{#1}{/bibtex/entry/command(bibtex:format)}
\xmlfilter{#1}{entry/command(bibtex:format)}
```

The `bibtex:bibtex` setup however, is using an indirect approach and only comes into action via the already mentioned `bibtex` setup. In that setup the `\xmlmain` command will expand the root element and when it sees the bibtex element, it will call the associated `bibtex:bibtex` setup. So here we need to call the `bibtex` setup.

```
\applytobibtexsession[somebibtex][bibtex]
```

Let's summarize what is needed to typeset a whole database:

```
\definebibtexsession  [somebibtex]
\registerbibtexfile   [somebibtex] [tugboat.bib]
\preparebibtexsession [somebibtex] [convert,strip]
\applytobibtexsession [somebibtex] [bibtex:list]
```

Here we use the predefined `bibtex:list` filter. Of course you need to define commands that are uses in the database.

8.3 The database

The xml database is quite simple and has the form (we omitted some fields):

```
<bibtex>
  <entry tag="hagen:tb19-3-311" category="article">
    <field name="number">3</field>
    <field name="bibdate">Fri Jul 13 10:24:20 MDT 2007</field>
    <field name="author">Hans Hagen</field>
    <field name="journal">TUGboat</field>
    <field name="title">{Visual Debugging in \TeX, Part 1: The Story}</field>
    <field name="ISSN">0896-3207</field>
```

```
    <field name="year">1998</field>
    <field name="pages">311--317</field>
    <field name="volume">19</field>
  </entry>
</bibtex>
```

It is good to keep in mind that we lowercase the name and category attributes.

By default there are no setups for the one character elements but if you need then you have to use the bibtex namespace, e.g.:

```
\startxmlsetups bibtex:field
    \xmlflushcontext{#1}
\stopxmlsetups
```

8.4 Sorting

maybe also per session

We can sort entries. For that we need to define a sort setup. First we create a sort vector based on some fields. The first argument (`bibtex`) is the sort vector.

```
\startxmlsetups bibtex:entry:getkeys
    \xmladdsortentry{bibtex}{#1}
        {\xmlfilter{#1}{/field[@name='author']/text()}}
    \xmladdsortentry{bibtex}{#1}
        {\xmlfilter{#1}{/field[@name='year'  ]/text()}}
    \xmladdsortentry{bibtex}{#1}
        {\xmlatt{#1}{tag}}
\stopxmlsetups
```

In the next setup we see this sorter being initialized. After that we filter some entries and add them to the to list of keys. Then we sort that list and flush it afterwards.

```
\startxmlsetups bibtex:entry:getkeys
    \xmladdsortentry{bibtex}{#1}
        {\xmlfilter{#1}{/field[@name='author']/text()}}
    \xmladdsortentry{bibtex}{#1}
        {\xmlfilter{#1}{/field[@name='year'  ]/text()}}
    \xmladdsortentry{bibtex}{#1}
        {\xmlatt{#1}{tag}}
\stopxmlsetups
```

The flusher simply shows some fields. You can do anything you want here with the content.

```
\startxmlsetups bibtex:entry:flush
    \xmlfilter{#1}{/field[@name='author']/context()} / %
    \xmlfilter{#1}{/field[@name='year'  ]/context()} / %
    \xmlatt{#1}{tag}\par
\stopxmlsetups
```

The setup that brings this all together is applied to the whole tree with the following command.

```
\xmlsetup{bibtex:somebibtex}{xml:bibtex:sorter}
```

The result is:

Don Knuth / 1984 / knuth:tb5-1-67
Donald E. Knuth / 1984 / knuth:tb5-1-4
Donald E. Knuth / 1984 / knuth:tb5-2-105
Donald E. Knuth / 1985 / knuth:tb6-1-36
Donald E. Knuth / 1986 / knuth:tb7-2-101
Donald E. Knuth / 1987 / knuth:tb8-2-135
Donald E. Knuth / 1987 / knuth:tb8-3-309
Donald E. Knuth / 1988 / knuth:tb9-2-152
Donald E. Knuth / 1989 / knuth:tb10-3-325
Donald E. Knuth / 1989 / knuth:tb10-4-529
Donald E. Knuth / 1990 / knuth:tb11-4-489
Donald E. Knuth / 1993 / knuth:tb14-4-387
Donald E. Knuth / 1996 / knuth:tb17-1-29
Donald Knuth and Pierre MacKay / 1987 / knuth:tb8-1-14
Donald Knuth / 1981 / knuth:tb2-3-5
Donald Knuth / 1982 / knuth:tb3-1-10
Donald Knuth / 1983 / knuth:tb4-2-64
Donald Knuth / 1986 / knuth:tb7-2-95
Donald Knuth / 1987 / knuth:tb8-1-6
Donald Knuth / 1987 / knuth:tb8-1-73
Donald Knuth / 1987 / knuth:tb8-2-210
Donald Knuth / 1987 / knuth:tb8-2-217
Donald Knuth / 1989 / knuth:tb10-1-8
Donald Knuth / 1989 / knuth:tb10-1-31
Donald Knuth / 1990 / knuth:tb11-1-13
Donald Knuth / 1990 / knuth:tb11-2-165
Donald Knuth / 1990 / knuth:tb11-4-497
Donald Knuth / 1990 / knuth:tb11-4-499

Donald Knuth / 1991 / knuth:tb12-2-313

You can call up the list of keys with

```
\xmlshowsorter{bibtex}
```

In our case this gives:

n	id	entry 1	entry 2	entry 3
1	324	Donald Knuth	1991	knuth:tb12-2-313
2	1397	Donald Knuth	1983	knuth:tb4-2-64
3	4173	Donald Knuth	1981	knuth:tb2-3-5
4	5247	Donald Knuth	1987	knuth:tb8-1-6
5	5943	Donald Knuth and Pierre MacKay	1987	knuth:tb8-1-14
6	7773	Donald Knuth	1989	knuth:tb10-1-8
7	10665	Donald Knuth	1990	knuth:tb11-1-13
8	10871	Donald Knuth	1986	knuth:tb7-2-95
9	11248	Donald E. Knuth	1990	knuth:tb11-4-489
10	12236	Donald Knuth	1990	knuth:tb11-4-497
11	12535	Donald E. Knuth	1984	knuth:tb5-2-105
12	12665	Donald E. Knuth	1993	knuth:tb14-4-387
13	12925	Donald E. Knuth	1988	knuth:tb9-2-152
14	14902	Donald E. Knuth	1987	knuth:tb8-2-135
15	15161	Donald Knuth	1987	knuth:tb8-2-217
16	21952	Donald Knuth	1990	knuth:tb11-4-499
17	23934	Donald Knuth	1987	knuth:tb8-2-210
18	24128	Donald Knuth	1987	knuth:tb8-1-73
19	26859	Donald E. Knuth	1989	knuth:tb10-4-529
20	28026	Donald Knuth	1989	knuth:tb10-1-31
21	28091	Donald E. Knuth	1996	knuth:tb17-1-29
22	28572	Donald E. Knuth	1986	knuth:tb7-2-101
23	28624	Donald E. Knuth	1984	knuth:tb5-1-4
24	31473	Donald E. Knuth	1985	knuth:tb6-1-36
25	34586	Donald E. Knuth	1987	knuth:tb8-3-309
26	34911	Don Knuth	1984	knuth:tb5-1-67
27	34950	Donald Knuth	1990	knuth:tb11-2-165
28	34976	Donald Knuth	1982	knuth:tb3-1-10
29	35196	Donald E. Knuth	1989	knuth:tb10-3-325

8.5 Encodings

It is a sure bet that many existing databases will use the traditional TEX accent

building commands. As in MkIV we live in an Unicode universe, such commands are translated into utf sequences when the database is loaded. When we pass the `convert` options to the preparation command, the entries will be cleaned up and accent commands will be replaced by proper utf sequences. This helps the sorter.

8.6 Messed up entries

As the bibTEX fields contains TEX code we need to process the content as TEX. This is why in the previous examples we applied the `context()` finalizer. The fact that we have TEX code means that such databases are rather bound to some macro package. For our purpose we had to define a few macros:

```
\startsetups bibtex-commands
    \def\MF  {MF}
    \def\MP  {MP}
    \def\TUB {TUGboat}
    \def\Mc  {Mac}
    \def\sltt{\tt}
    \let\acro\firstofoneargument
\stopsetups
```

You can best do this grouped is that there is no interference with existing code. You can collect definitions in a setup or buffer and flush that one inside the group.

However, we also provide another method. A second argument to the preparation command gives options. Examples of options are `convert`, which converts entries to proper utf, and `strip` which converts commands and strips redundant braces.

```
\preparebibtexsession [somebibtex] [convert,strip]
```

All commands are mapped onto `\bibtexcommand` which defaults to using predefined local commands. You predefine such a local command with:

```
\defbibtexcommand\MF    {MF}
\defbibtexcommand\MP    {MP}
\defbibtexcommand\TUB   {TUGboat}
\defbibtexcommand\Mc    {Mac}
\defbibtexcommand\sltt  {\tt}
\defbibtexcommand\acro#1{#1}
```

If you use a database like `tugboat.bib` you will need quite some more defini-

tions. Unknown commands are reported on the console. When a command is available in ConTEXt it will be used unless a specific one is defined.

Here's a setup that shows what goes on inside:

```
\startxmlsetups bibtex:show
    \xmlshow{#1}
\stopxmlsetups

\applytobibtexsession[somebibtex][bibtex:show]
```

8.7 Traditional usage

In this section we will describe how you can use this approach as a drop in for the traditional, pure TEX based one.

9 Building paragraphs

9.1 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 paragrphs into lines I never really looked in 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 simular to expansion level one support 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 looks 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’ appeoach.

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

9.2 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 makes 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 trasnslated 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 spend 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.

9.3 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[12] 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 somewhat strange. When a kern of subtype zero is seen, and when 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

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

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:[13]

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 are 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 if that we really have to wonder what certain code does and if it's still needed.

9.4 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)
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 soem code can be omitted,

[13] 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.

The reason why the protruding is relative 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 provides once that 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.

9.5 Usage

The basic parbuilder is enabled and disabled as follows:[14]

```
\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, 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.

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

9.6 Conclusion

The module started working quiet 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 do. For instance the large functions need to be broken into smaller steps. And of course more testing is needed.

10 Tagged PDF

10.1 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, maybe except for accessibility. Another reason for not having spent too much time on it before is that the specification was not that inviting.

Anyhow, when I saw Ross Moore[15] 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, not the low level ConTEXt code.

And of course implementing it should not take more than a few days work, certainly not in an exceptional 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 happing under the hood. Unfortunately it is quite hard to find a proper 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 loud. But you cannot browse the structure in this free program 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 feature is not enabled by default.

[15] 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.

10.2 Implementation

So, what does tagging boil down to? We can best look at how tagged information is shown in Acrobat. Figure 10.1 shows the content tree that has been added (automatically) to a document while figure 10.2 shows a different view.

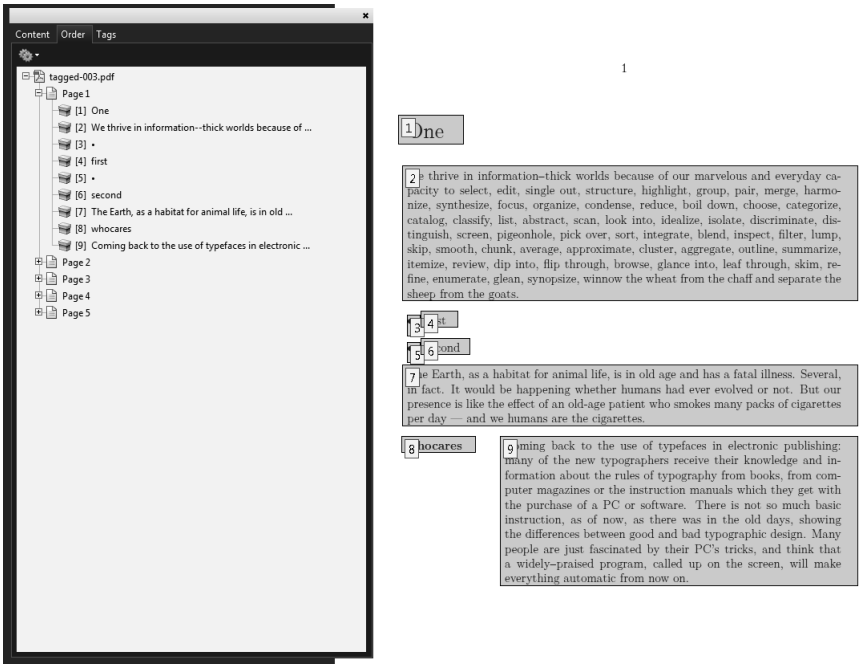


Figure 10.2 Acrobat showing the tag order.

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

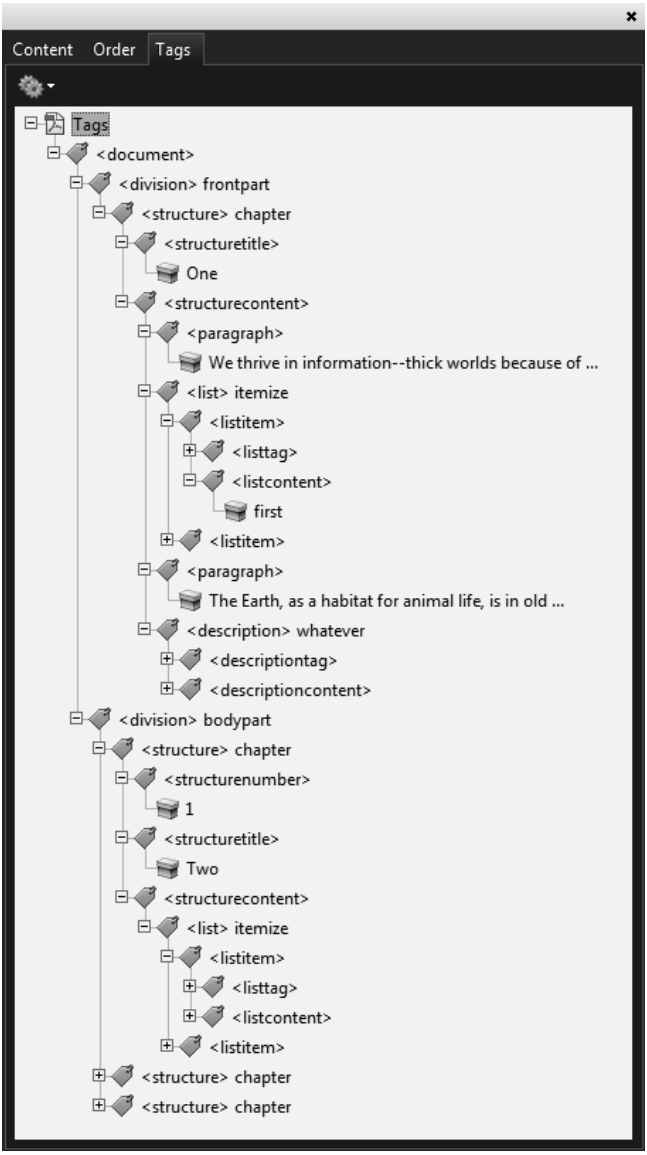


Figure 10.1 A tag list in Acrobat.

This resulted in tagging framework that uses explicit tags. In that 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
```

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

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

\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 to use 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 tagging information at the TEX end. The backend drivers 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.

construct (Span) delimited (Quote) delimitedblock (BlockQuote) description (Div) descriptioncontent (Div) descriptionsymbol (Span) descriptiontag (Div) division (Div) document (Div) float (Div) floatcaption (Caption) floatcontent (P) floattag (Span) floattext (Span) formula (Div) formulacontent (P) formulaset (Div) formulatag (Span) image (P) item (Li) itemcontent (LBody) itemgroup (L) itemtag (Lbl) link (Link) list (TOC) listcontent (P) listdata (P) listitem (TOCI) listpage (Reference) listtag (Lbl) margintext (Span) margintextblock (Span) math (Div) merror (Span) mfrac (Span) mi (Span) mn (Span) mo (Span) mover (Span) mpgraphic (P) mroot (Span) mrow (Span) ms (Span) msqrt (Span) msub (Span) msubsup (Span) msup (Span) mtext (Span) munder (Span) munderover (Span) paragraph (P) register (Div) registerentries (Div) registerentry (Span) registerpage (Span) registerpages (Span) registersection (Div) registersee (Span) regis-

tertag (Span) section (Sect) sectioncontent (Div) sectionnumber (H) sectiontitle (H) subformula (Div) subsentence (Span) table (Table) tablecell (TD) tablerow (TR) tabulate (Table) tabulatecell (TD) tabulaterow (TR) verbatim (Code) verbatimblock (Code) verbatimline (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}
```

You can use the * as wildcard. The elements are separated by a -. 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.

10.3 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 spaces 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.[16] But anyhow we could 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)
```

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

```
                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 compensate 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 as 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 inline mathml. Figure 10.3 demonstrates that a first approach does not even look that bad. In future versions we might deal with table like math constructs, like matrices.

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 example at the end.

10.4 Conclusion

Surprisingly implementing all this didn't take that much work. Of course detailed automated structure support from the complete ConTEXt kernel will take

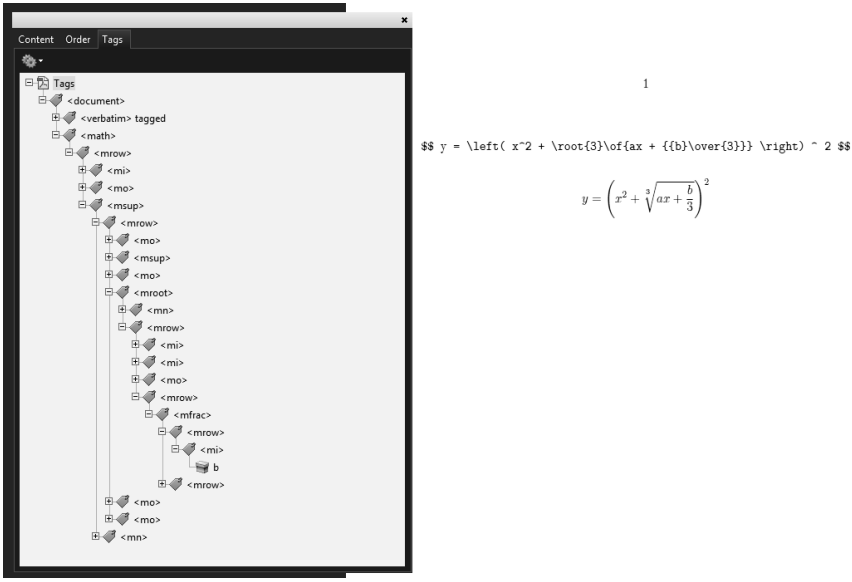


Figure 10.3 Experimental math tagging.

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 extend alternate representations and alternate texts will be supported. Experiments with the reading loud 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 surface buglets and oversights.

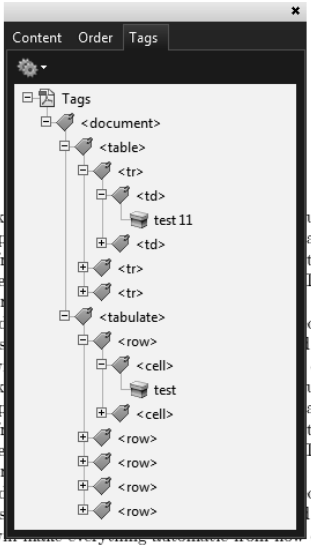
10.5 Some more examples

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 10.4. Some examples of tables are shown in figure 10.5. Future versions will have a bit more structure. Tables of contents (see figure 10.6) and registers (see figure 10.7) are also tagged. One might wonder what the use is of this. In Figure 10.8 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 end up in the structure tree, but in the typeset document they move around (normally forward when there is no room).

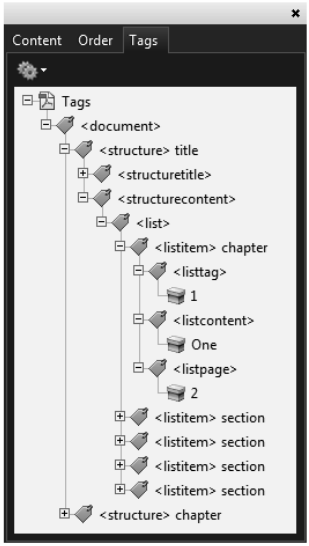
Figure 10.4 Verbatim, including dedicated instances.



test 11	test 12
test 21	test 22
test 33	

test Coming back ublishing: many of the
new typograp ation about the rules of
typography fr the instruction manuals
which they ge There is not so much ba-
sic instructio showing the differences
between good ople are just fascinated
by their PC's l program, called up on
the screen, w on.
test Coming back ublishing: many of the
new typograp ation about the rules of
typography fr the instruction manuals
which they ge There is not so much ba-
sic instructio showing the differences
between good ople are just fascinated
by their PC's l program, called up on
the screen, w on.

Figure 10.5 Natural tables as well as the tabulate mechanism is supported.



Contents

1 One 2
1.1 alpha 2
1.2 beta 2
1.3 gamma 2
1.4 delta 2

Figure 10.6 Tables of content with specific entries tagged.

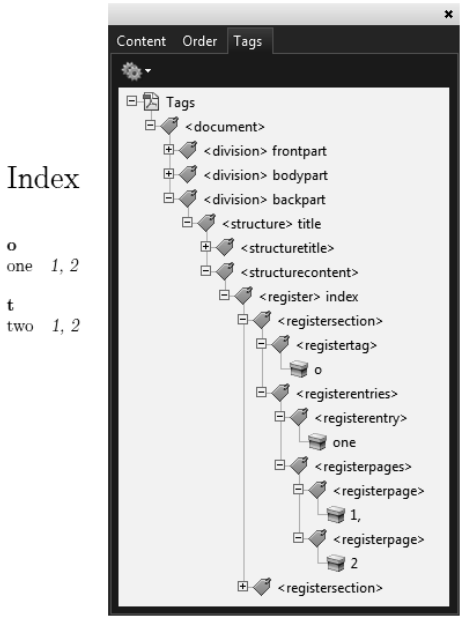


Figure 10.7 A detailed view of registered is provided.

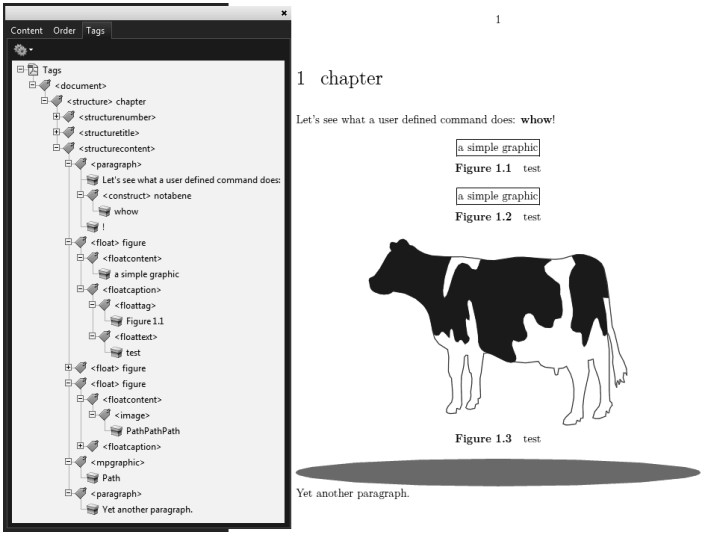


Figure 10.8 Floats tags end up in text stream. Watch the user defined construct.

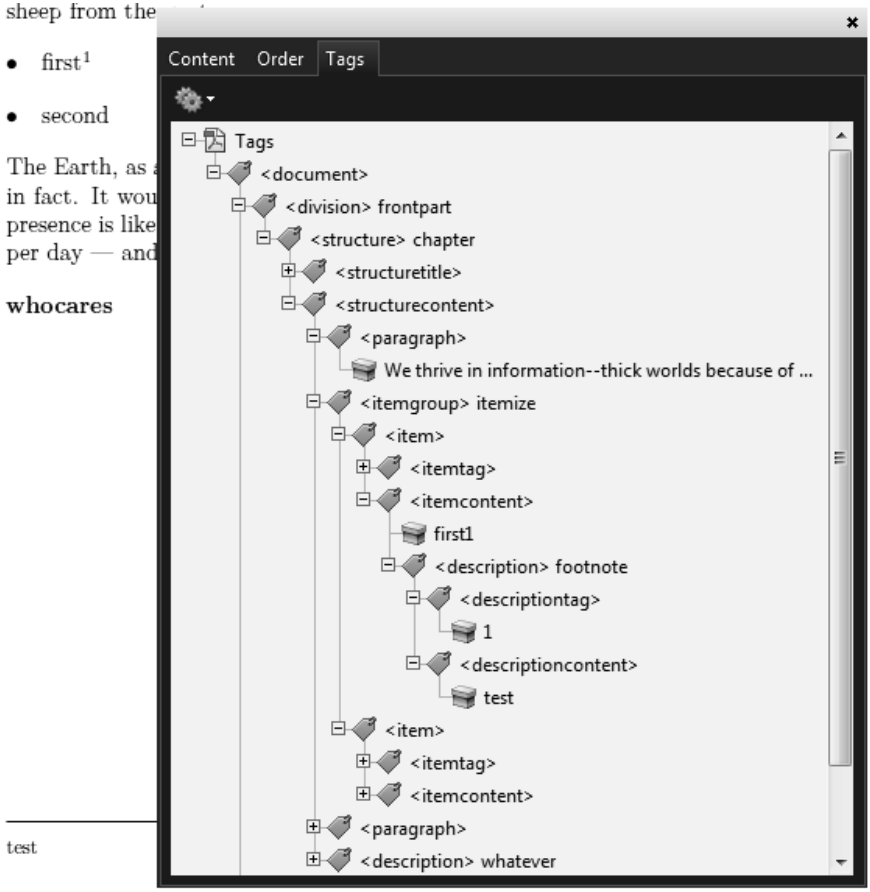


Figure 10.9 Footnotes are shown at the place in the input (flow).

11 Including pages

11.1 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.

11.2 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 does not want to deal with. A pdf file is in fact just a collections 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.

11.3 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 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 Lua-TEX 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 (like on tables 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 that 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.

11.4 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.

12 Exporting XML

12.1 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.

12.2 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 environment 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 that route.

An alternative is to use `tex4ht` which does an impressive job for LaTeX, and supports ConTEXt to some extend 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 verbose structured so let's have a look at that.

12.3 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:[17]

```
<?xml standalone='yes' encoding='utf-8' ?>

<!-- input filename   : exported-001      -->
<!-- processing date  : 09/08/10 01:00:22 -->
<!-- context version  : 2010.09.05 16:30  -->
<!-- exporter version : 0.10              -->

<document language='en'>
  <section detail='chapter'>
    <sectionnumber>1</sectionnumber>
    <sectiontitle>Test</sectiontitle>
    <sectioncontent>
A paragraph.
      <break/>
 Another paragraph.
    </sectioncontent>
  </section>
</document>
```

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

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

\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.

```
<document language='en'>
  <table>
    <tablerow>
      <tablecell align='flushleft'>test 1.1 </tablecell>
      <tablecell align='flushleft' rows='2'>test 1.2 </tablecell>
    </tablerow>
    <tablerow>
      <tablecell align='flushleft'>test 2.1 </tablecell>
    </tablerow>
    <tablerow>
      <tablecell align='flushleft'>test 3.1 </tablecell>
      <tablecell align='flushleft'>test 3.2 </tablecell>
    </tablerow>
    <tablerow>
      <tablecell align='flushleft'>test 4.1 </tablecell>
      <tablecell></tablecell>
    </tablerow>
    <tablerow>
      <tablecell align='flushright' columns='2'>test 5.1 </tablecell>
    </tablerow>
  </table>
</document>
```

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 paragraph in a tabulate becomes a cell, we need to reconstruct the paragraphs.

```
\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:

```
<document language='en'>
  <tabulate>
    <tabulaterow>
      <tabulatecell align='flushleft'>zero</tabulatecell>
      <tabulatecell>line one
      <break/>
line two
        <break/>
 line three</tabulatecell>
      <tabulatecell align='flushright'>0</tabulatecell>
    </tabulaterow>
    <tabulaterow>
      <tabulatecell align='flushleft'>two</tabulatecell>
      <tabulatecell>before <verbatim>connect</verbatim>
      <break/>
after</tabulatecell>
      <tabulatecell align='flushright'>2</tabulatecell>
    </tabulaterow>
    <tabulaterow>
      <tabulatecell align='flushleft'>three</tabulatecell>
      <tabulatecell>before <verbatim>connect</verbatim> after</tabulatecell>
      <tabulatecell align='flushright'>3</tabulatecell>
    </tabulaterow>
    <tabulaterow>
      <tabulatecell align='flushleft'>four</tabulatecell>
      <tabulatecell>before inbetween
        <break/>
```

```
 after</tabulatecell>
      <tabulatecell align='flushright'>4</tabulatecell>
    </tabulaterow>
  </tabulate>
</document>
```

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.

12.4 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.

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[18]. 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:

[18] We will see if this tree will be used for other purposes in the future.

```
<document language='en'>
  <itemgroup detail='itemize' symbol='n'>
    <item>
      <itemtag>1.</itemtag>
      <itemcontent>one</itemcontent>
    </item>
    <item>
      <itemtag>2.</itemtag>
      <itemcontent>two</itemcontent>
    </item>
  </itemgroup>
  <itemgroup detail='itemize' packed='yes' symbol='a'>
    <item>
      <itemtag>a.</itemtag>
      <itemcontent><delimited detail='quote'>‘one’</delimited></itemcontent>
    </item>
    <item>
      <itemtag>b.</itemtag>
      <itemcontent><delimited detail='quote'>‘two’</delimited></itemcontent>
    </item>
  </itemgroup>
</document>
```

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 carries the attribute.[19]

I already mentioned collapsing. As paragraphs in a tabulate get split in cells, we encounter a mixture that cannot be flushed sequentially. However, as each cell is tagged unique 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.

[19] Only glyph nodes are investigated for structure.

```
\starttext

\startchapter[title=Notes]

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

\stopchapter

\stoptext
```

Currently this will end up as follows:

```
<document language='en'>
  <section detail='chapter'>
    <sectionnumber>1</sectionnumber>
    <sectiontitle>Notes</sectiontitle>
    <sectioncontent>
test<descriptionsymbol detail='footnote' insert='1'>1</descriptionsymbol>
test<descriptionsymbol detail='footnote' insert='2'>2</descriptionsymbol>
      <description detail='footnote'>
        <descriptiontag insert='1'>1 </descriptiontag>
        <descriptioncontent>note a</descriptioncontent>
      </description>
      <description detail='footnote'>
        <descriptiontag insert='2'>2 </descriptiontag>
        <descriptioncontent>note b</descriptioncontent>
      </description>
    </sectioncontent>
  </section>
</document>
```

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.

```
<document language='en'>
  <float detail='figure' reference='fig:cow'>
    <floatcontent><image name='cow.pdf'></image></floatcontent>
    <floatcaption>
      <floattag>Figure 1</floattag>
      <floattext>It looks like a cow.</floattext>
    </floatcaption>
  </float>
</document>
```

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 the ConTEXt specific way, so no interpretation takes place.

```
<document language='en'>
  <section detail='chapter' reference='alpha'>
    <sectionnumber>1</sectionnumber>
    <sectiontitle>One</sectiontitle>
    <sectioncontent>
In <link reference='beta' location='aut:2'>chapter 2</link> ...
    </sectioncontent>
  </section>
```

```
  <section detail='chapter' reference='beta'>
    <sectionnumber>2</sectionnumber>
    <sectiontitle>Two</sectiontitle>
    <sectioncontent>
In <link reference='alpha' location='aut:1'>chapter 1</link> ...
    </sectioncontent>
  </section>
</document>
```

As ConTEXt has an integrated referencing system that deals with internal as well as external references, url's, special interactive actions like controlling wigets 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.

```
<document language='en'>
  <paragraph>You can visit <link reference='url(www.pragma-ade.com)' url='www.pra
  <paragraph>You can visit <link reference='url(pragma)' url='www.pragma-ade.com'
</document>
```

12.5 Math

Of course there are limitations. For instance TEXies doing math might wonder if we can export formulas. To some extend 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:

```
<document language='en'>
Is it
  <math>
    <mrow>
      <mi></mi>
      <mo>=</mo>
      <mi></mi>
      <msup>
        <mi></mi>
        <mn>2</mn>
      </msup>
    </mrow>
  </math>
 maybe:
  <formula>
    <formulacontent>
      <math>
        <mrow>
          <mi> </mi>
          <mo>=</mo>
          <mrow>
            <mfrac>
              <mrow>
                <mrow>
                  <mo>√ </mo>
                  <mroot>
                    <mi></mi>
                  </mroot>
                </mrow>
              </mrow>
              <mrow>
```

```
                <mi> </mi>
              </mrow>
            </mfrac>
          </mrow>
        </mrow>
      </math>
    </formulacontent>
  </formula>
</document>
```

More complex math (like matrices) will be dealt with in due time as for this and Aditya and I have to take tagging into account when we revision 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:

```
<document language='en'>
  <verbatimblock detail='typing'>
    <verbatimline>
line one
    </verbatimline>
    <verbatimline>
line two
    </verbatimline>
  </verbatimblock>
</document>
```

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:

```
<document language='en'>
  <paragraph><margintextblock detail='left'>left 1</margintextblock> test
test</paragraph>
  <paragraph>test test</paragraph>
  <paragraph><margintext detail='inrightmargin'> right 1</margintext>
</paragraph></document>
```

However, this might change in future versions.

12.6 Formatting

The output is somewhat formatted. 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:

```
<document language='en'>
  <display>
<inline>test <display> test test</display></inline>
  </display>
</document>
```

Keep in mind that elements have no influence on the typeset result apart from introducing spaces when used 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 change ending up with those megabyte long one–liners that some applications produce.

12.7 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:

```
<document language='en'>
  <section detail='section'>
    <sectionnumber>1</sectionnumber>
    <sectiontitle>Normal section</sectiontitle>
    <sectioncontent>
normal
    </sectioncontent>
  </section>
  <section detail='specialsection'>
    <sectionnumber>2</sectionnumber>
    <sectiontitle>Special section</sectiontitle>
    <sectioncontent>
special
    </sectioncontent>
  </section>
</document>
```

12.8 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 has 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.