The language mix

Abstract
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.

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.

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.

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 = "bluegreen",
    foregroundcolor = "white",
  }
  local basespec = {
    framecolor = "bluered",
    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 optimizations 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.

84	40	78	80	91	20	34	77	28	55	48	63	37	51	95	91	63	72	15	61
2	25	14	80	16	40	13	11	99	22	51	84	61	30	64	52	49	97	29	77
53	77	40	89	29	35	80	91	7	94	53	9	20	66	89	35	7	2	46	7
24	97	90	85	27	54	38	76	51	67	53	4	44	93	93	72	29	74	64	36
69	17	44	88	83	33	23	89	35	68	95	59	66	86	44	92	40	81	68	91
48	22	95	92	15	88	64	43	62	28	78	31	45	23	19	28	56	42	17	90
11	13	50	76	98	93	68	38	75	37	30	23	58	25	16	73	13	79	17	74
8	95	6	52	18	24	79	73	65	96	64	76	10	14	52	8	7	21	46	82
57	75	6	16	99	21	89	13	99	6	87	8	1	92	59	18	17	39	91	82
36	55	58	45	69	10	53	75	31	99	58	87	75	63	4	75	83	92	87	83

Figure 1. The result of the displayed Lua code.

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`.[1]

```
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.

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.[2]

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
```

Figure 2. An example of using the font tester.

```
  \endgraf
  score: \xmlfunction{#1}{totalscore}
  \blank
\stopxmlsetups

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

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.

Tools

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

Take this:

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

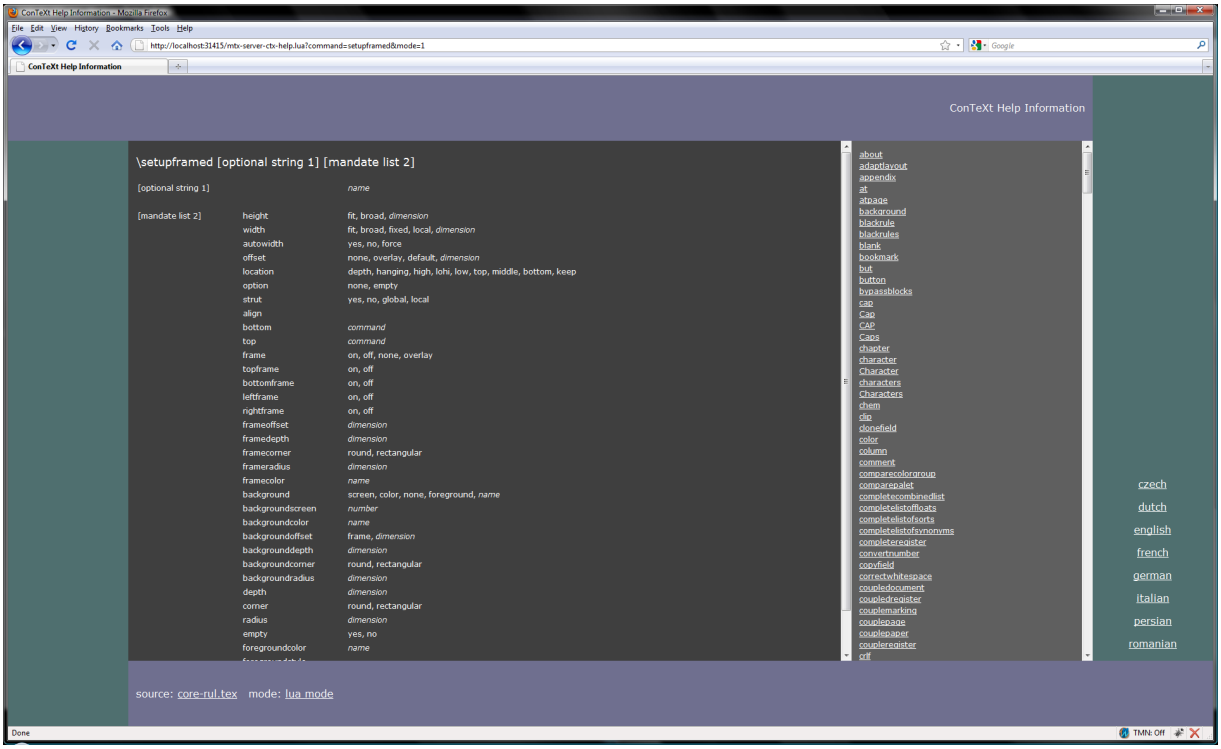

Figure 3. An example of a help screen for a command.

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.[3]

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 2 you seen an example of what the font testing tool does.

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

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

So again we have a hybrid usage although in this case the user is not confronted with Lua and/or TEX at all. The same is true for the other goodie, shown in figure 3. Actually, such a goodie has always been part of the ConTEXt distribution but it has been rewritten in Lua.

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.

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.

Notes

1. Similar methods exist for processing XML files.
2. 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.
3. This application is not intentional but just a side effect.

Hans Hagen
Pragma ADE, Hasselt
pragma (at) wxs dot nl