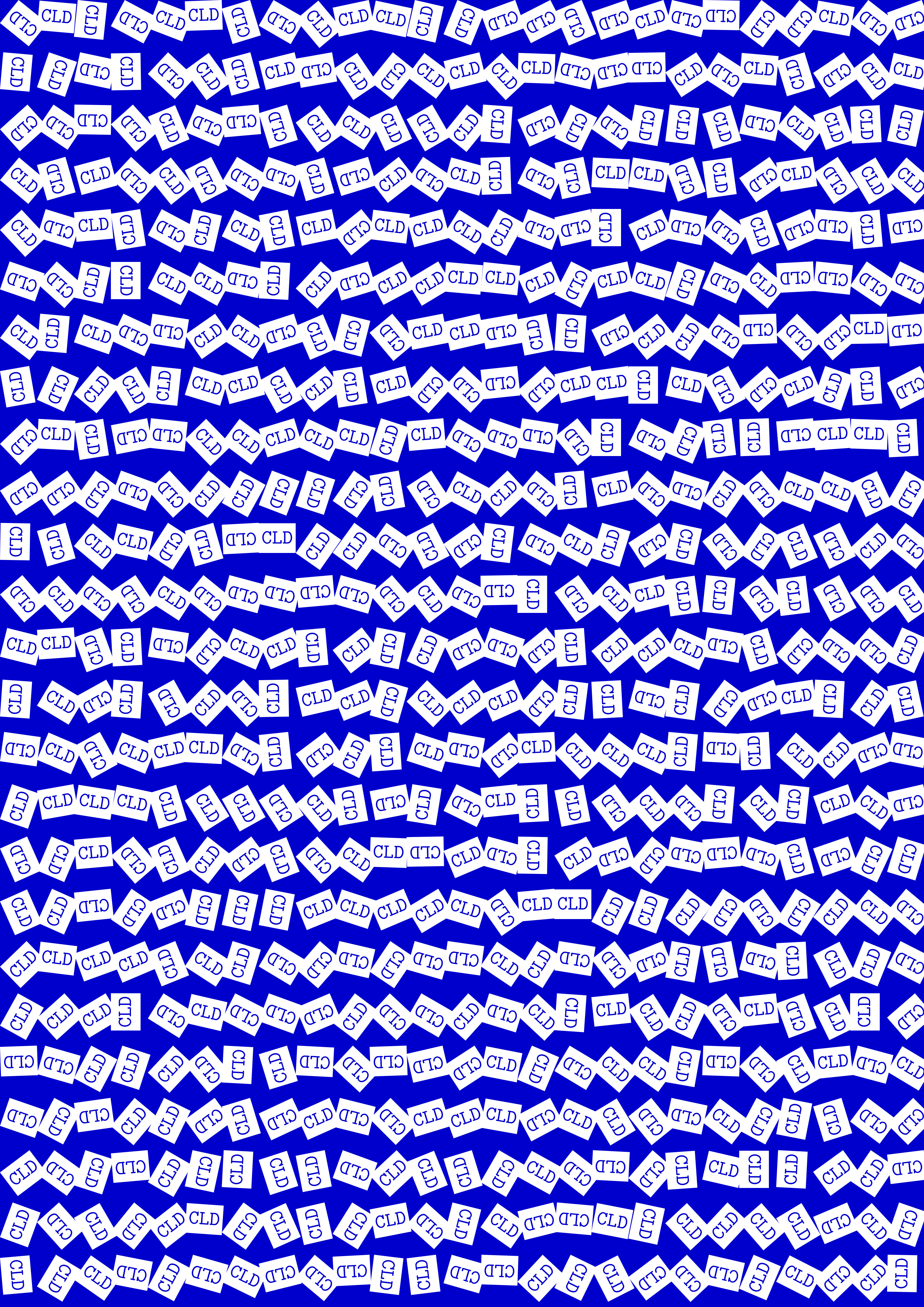
CLD

Introduction

Sometimes you hear folks complain about the TEX input language, i.e. the backslashed commands that determine your output. Of course, when alternatives are being discussed every one has a favourite programming language and of course in practice coding a document in each of them triggers similar sentiments with regards to coding as TEX itself does.

However, just for fun, I added a couple of commands to ConTEXt MkIV that permit you to code your document in Lua. After all it is surprisingly simple to implement a feature like this due to metatables. I was wondering if there was a more natural way to deal with commands at the Lua end. Of course it's a bit slower but often more readable when mixed with Lua code.

So, we now can code in TEX, xml, METAPOST, as well as in Lua. Coding in Lua makes a lot of sense when you generate content, for instance from a database.

From the users perspective a ConTEXt run goes like:

```
context yourfile
```

and by default a file with suffix `tex` will be processed. There are however a few other options:

```
context yourfile.xml
context yourfile.rlx --forcexml
context yourfile.lua
context yourfile.pqr --forcelua
context yourfile.cld
context yourfile.xyz --forcecld
```

When processing a Lua file the given file is loaded and just processed. This options will seldom be used as it is way more efficient to let `mtxrun` process that file. However, the last two variants are what we will discuss here. The suffix `cld` is a shortcut for ConTEXt Lua Document.

A simple `cld` file looks like this:

```
context.starttext()
context.chapter("Hello There!")
context.stoptext()
```

So, yes, you need to know the ConTEXt commands in order to use this mechanism. In spite of what you might expect, the codebase involved in this gimmick is not that large. If you know ConTEXt, and if you know how to call commands, you basically can use this Lua method.

There are a few rules that you need to be aware of. First of all no syntax checking is done. Second you need to know what the given commands expects in terms of arguments. Third, the type of your arguments matter:

`nothing`	:	just the command, no arguments
`string`	:	argument with curly braces

`array`	: list between square backets
`hash`	: assignment list between square brackets

In the code above you have seen examples of this but here are some more:

```
context.chapter("Some title")
context.chapter({ "first" }, "Some title")
context.startchapter({ title = "Some title", label = "first" })
```

This is equivalent to:

```
\chapter{Some title}
\chapter[first]{Some title}
\startchapter[title={Some title},label=first]
```

Strings are interpreted as TEX input, so:

```
context.mathematics("\\sqrt{2^3}")
```

or, if you don't want to escape:

```
context.mathematics([[\sqrt{2^3}]])
```

is okay. As TEX math is a language in its own and a de-facto standard way of inputting math this is quite natural, even at the Lua end.

Appetizer

Before we give some more examples, we will have a look at the way the title page is made:

```
local todimen = number.todimen

context.startTEXpage()

local paperwidth  = tex.dimen.paperwidth
local paperheight = tex.dimen.paperheight
local nofsteps    = 25
local firstcolor  = "darkblue"
local secondcolor = "white"

context.definelayer(
    { "titlepage" }
)

context.setuplayer(
    { "titlepage" },
    {
        width  = todimen(paperwidth),
        height = todimen(paperheight),
    }
)

context.setlayerframed(
    { "titlepage" },
    { offset = "-5pt" },
    {
        width  = todimen(paperwidth),
        height = todimen(paperheight),
        background = "color",
        backgroundcolor = firstcolor,
        backgroundoffset = "10pt",
        frame = "off",
    },
    ""
)

for i=1, nofsteps do
    for j=1, nofsteps do
        context.setlayerframed(
            { "titlepage" },
            {
                x = todimen((i-1) * paperwidth /nofsteps),
                y = todimen((j-1) * paperheight/nofsteps),
                rotation = math.random(360),
            },
            {
                frame = "off",
                background = "color",
                backgroundcolor = secondcolor,
                foregroundcolor = firstcolor,
                foregroundstyle = "type",
            },
            "CLD"
        )
    end
end

context.tightlayer(
    { "titlepage" }
)

context.stopTEXpage()

return true
```

This does not look that bad, does it? Of course in pure TEX code it looks mostly the same but loops and calculations look a bit more natural in Lua then in TEX.

A few examples

As it makes most sense to use the Lua interface for generated text, here is another example with a loop:

```
context.startitemize({ "packed" })
  for i=1,10 do
    context.startitem()
      context("this is item %i",i)
    context.stopitem()
  end
context.stopitemize()
```

Just as you can mix TEX with xml and METAPOST, you can define bits and pieces of a document in Lua. Tables are good candidates:

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

Here we see a function call to `context` in the most indented line. The first argument is a format and the rest of the arguments is substituted into this format. The result is shown in table 1. The line correction is ignored when we use this table as a float, otherwise it assures proper vertical spacing around the table.

Not all code will look as simple as this. Consider the following:

85	18	46	35	67	60	87	95	34	3	71	7	43	47	15	73	66	81	93	9
75	89	8	15	33	42	25	92	57	58	10	31	83	93	69	35	72	65	98	41
3	49	85	72	88	79	48	89	62	70	50	94	67	99	99	25	89	72	38	12
87	94	53	39	39	95	16	69	72	3	62	72	85	33	46	35	39	91	48	77
23	30	6	50	60	88	21	51	83	38	17	33	26	66	16	24	59	30	86	55
34	92	47	11	45	90	66	76	84	44	60	31	91	52	13	39	84	82	17	85
64	7	76	30	74	88	70	97	27	71	88	4	56	69	72	17	25	23	64	64
39	14	80	83	92	91	29	15	4	24	83	58	64	14	38	67	65	97	38	16
5	58	28	13	62	83	13	92	41	99	63	59	48	92	59	9	26	70	14	25
5	70	74	43	91	61	25	1	55	41	96	72	32	94	17	75	6	1	97	82

Table 1 A table generated by Lua.

```
context.placefigure(
  "caption",
  function() context.externalfigure( { "cow.pdf" } ) end
)
```

Here we pass an argument wrapped in a function. If we would not do that, the external figure would end up wrong, as arguments to functions are evaluated before the function that gets them. A function argument is treated special and in this case the external figure ends up right. Here is another example:

```
\startluacode
  context.placefigure("Two cows!",function()
    context.bTABLE()
      context.bTR()
        context.bTD()
          context.externalfigure(
              { "cow.pdf" },
              { width = "3cm", height = "3cm" }
          )
        context.eTD()
        context.bTD({ align = "{lohi,middle}" } )
          context("and")
        context.eTD()
        context.bTD()
          context.externalfigure(
              { "cow.pdf" },
              { width = "4cm", height = "3cm" }
          )
        context.eTD()
      context.eTR()
    context.eTABLE()
  end)
\stopluacode
```

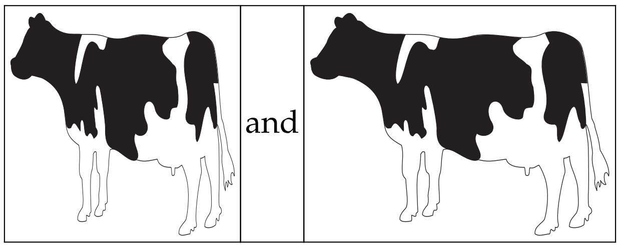

Figure 1 Two cows!

In previous examples the function has no return value and as such ends up as string, but other types are also possible. The following two calls are equivalent:

```
context.chapter(
    { "ref" },
    "Title"
)

context.chapter(
    function() return { "ref" } end,
    function() return "Title" end
)
```

and both are effectively:

```
\chapter[ref]{Title}
```

Because the ConTEXt user interface is quite consistent this kind of tricks is possible. Of course more obscure interfaces can be supported as well by returning a function.

```
context.chapter(function return "*", "direct" end, "Title")
```

Of course, this also works out well then:

```
tex.sprint(ctx.catcodes,"\\chapter*{Title}")
```

But ConTEXt is not to happy with such chapters. The `direct` signals that no braces or brackets should be added to the `*`.

A function call to `context` acts like a print, as in:

```
\startluacode
  context("test ")
  context.bold("me")
  context(" first")
\stopluacode
```

However, internally we use the the `string.format` function so you can pass more arguments.

```
\startluacode
  context.startimath()
  context("%s = %0.5f",utf.char(0x03C0),math.pi)
  context.stopimath()
\stopluacode
```

Special commands

There is one function in the `context` namespace that is no macro:

```
context.runfile("somefile.cld")
```

Another useful command is:

```
context.enabletrackers( { "cld.print" } )
```

but this is just the equivalent of the macro with the same name:

```
\enabletrackers[cld.print]
```

Disclaimer

This mechanism is still experimental and might change a bit as I'm not entirely convinced that this is the right way to do things.

Hans Hagen
Hasselt NL
July 2009