Zpravodaj Československého sdružení uživatelů TeXu

Hans Hagen

ConTeXt - Lua Documents

Zpravodaj Československého sdružení uživatelů TeXu, Vol. 27 (2017), No. 1-2, 3–54

Persistent URL: http://dml.cz/dmlcz/150267

ConTEXt–Lua Documents

Hans Hagen

In ConTEXt, it is now possible to prepare documents in a mixture of TEX, xml, MetaPost, and Lua. The article gives a short introduction into the programming language of Lua and then goes on to describe how Lua can be used for programming in ConTEXt MkIV.

Keywords: Lua, LuaTEX, ConTEXt, MkIV

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. In practice coding a document in each of them triggers similar sentiments with regards to coding as TEX itself does.

So, just for fun, I added a couple of commands to ConTEXt MkIV that permit coding a document in Lua. In retrospect it has been surprisingly easy to implement a feature like this using metatables. Of course it's a bit slower than using TEX as input language but sometimes the Lua interface is more readable given the problem at hand.

After a while I decided to use that interface in non-critical core ConTEXt code and in styles (modules) and solutions for projects. Using the Lua approach is sometimes more convenient, especially if the code mostly manipulates data. For instance, if you process xml files of database output you can use the interface that is available at the TEX end, or you can use Lua code to do the work, or you can use a combination. So, from now on, in ConTEXt you can code your style and document source in (a mixture of) TEX, xml, MetaPost and in Lua.

In the following pages I will introduce typesetting in Lua, but as we rely on ConTEXt it is unavoidable that some regular ConTEXt code shows up. The fact that you can ignore backslashes does not mean that you can do without knowledge of the underlying system. I expect that the user is somewhat familiar with this macro package. Some chapters are follow ups on articles or earlier publications.

Although much of the code is still experimental it is also rather stable. Some helpers might disappear when the main functions become more clever. This manual is definitely far from complete. If you find errors, please let me know. If you think that something is missing, you can try to convince me to add it.

doi: 10.5300/2017-1-2/3

A bit of Lua

The language

Small is beautiful and this is definitely true for the programming language Lua (moon in Portuguese). We had good reasons for using this language in LuaTeX: simplicity, speed, syntax and size to mention a few. Of course personal taste also played a role and after using a couple of scripting languages extensively the switch to Lua was rather pleasant.

As the Lua reference manual is an excellent book there is no reason to discuss the language in great detail: just buy 'Programming in Lua' by the Lua team. Nevertheless I will give a short summary of the important concepts but consult the book if you want more details.

Data types

The most basic data type is `nil`. When we define a variable, we don't need to give it a value:

```
local v
```

Here the variable `v` can get any value but till that happens it equals `nil`. There are simple data types like `numbers`, `booleans` and `strings`. Here are some numbers:

```
local n = 1 + 2 * 3
local x = 2.3
```

Numbers are always floats[1] and you can use the normal arithmetic operators on them as well as functions defined in the math library. Inside TeX we have only integers, although for instance dimensions can be specified in points using floats but that's more syntactic sugar. One reason for using integers in TeX has been that this was the only way to guarantee portability across platforms. However, we're 30 years along the road and in Lua the floats are implemented identical across platforms, so we don't need to worry about compatibility.

Strings in Lua can be given between quotes or can be so called long strings forced by square brackets.

```
local s = "Whatever"
local t = s .. ' you want'
local u = t .. [[ to know]] .. [[--[ about Lua!]--]]
```

The two periods indicate a concatenation. Strings are hashed, so when you say:

```
local s = "Whatever"
local t = "Whatever"
local u = t
```

[1] This is true for all versions upto 5.2 but following version can have a more hybrid model.

only one instance of `Whatever` is present in memory and this fact makes LUA very efficient with respect to strings. Strings are constants and therefore when you change variable `s`, variable `t` keeps its value. When you compare strings, in fact you compare pointers, a method that is really fast. This compensates the time spent on hashing pretty well.

Booleans are normally used to keep a state or the result from an expression.

```
local b = false
local c = n > 10 and s == "whatever"
```

The other value is `true`. There is something that you need to keep in mind when you do testing on variables that are yet unset.

```
local b = false
local n
```

The following applies when `b` and `n` are defined this way:

```
b == false  true
n == false  false
n == nil    true
b == nil    false
b == n      false
```

Often a test looks like:

```
if somevar then
    ...
else
    ...
end
```

In this case we enter the else branch when `somevar` is either `nil` or `false`. It also means that by looking at the code we cannot beforehand conclude that `somevar` equals `true` or something else. If you want to really distinguish between the two cases you can be more explicit:

```
if somevar == nil then
    ...
elseif somevar == false then
    ...
else
    ...
end
```

or

```
if somevar == true then
    ...
else
```

```
    ...
end
```

but such an explicit test is seldom needed.

There are a few more data types: tables and functions. Tables are very important and you can recognize them by the same curly braces that make TEX famous:

```
local t = { 1, 2, 3 }
local u = { a = 4, b = 9, c = 16 }
local v = { [1] = "a", [3] = "2", [4] = false }
local w = { 1, 2, 3, a = 4, b = 9, c = 16 }
```

The `t` is an indexed table and `u` a hashed table. Because the second slot is empty, table `v` is partially indexed (slot 1) and partially hashed (the others). There is a gray area there, for instance, what happens when you nil a slot in an indexed table? In practice you will not run into problems as you will either use a hashed table, or an indexed table (with no holes), so table `w` is not uncommon.

We mentioned that strings are in fact shared (hashed) but that an assignment of a string to a variable makes that variable behave like a constant. Contrary to that, when you assign a table, and then copy that variable, both variables can be used to change the table. Take this:

```
local t = { 1, 2, 3 }
local u = t
```

We can change the content of the table as follows:

```
t[1], t[3] = t[3], t[1]
```

Here we swap two cells. This is an example of a parallel assigment. However, the following does the same:

```
t[1], t[3] = u[3], u[1]
```

After this, both `t` and `u` still share the same table. This kind of behaviour is quite natural. Keep in mind that expressions are evaluated first, so

```
t[#t+1], t[#t+1] = 23, 45
```

makes no sense, as the values end up in the same slot. There is no gain in speed so using parallel assignments is mostly a convenience feature.

There are a few specialized data types in LUA, like `coroutines` (built in), `file` (when opened), `lpeg` (only when this library is linked in or loaded). These are called ‘userdata’ objects and in LUATEX we have more userdata objects as we will see in later chapters. Of them nodes are the most noticeable: they are the core data type of the TEX machinery. Other libraries, like `math` and `bit32` are just collections of functions operating on numbers.

Functions look like this:

```
function sum(a,b)
  print(a, b, a + b)
```

```
end
```

or this:

```
function sum(a,b)
  return a + b
end
```

There can be many arguments of all kind of types and there can be multiple return values. A function is a real type, so you can say:

```
local f = function(s) print("the value is: " .. s) end
```

In all these examples we defined variables as `local`. This is a good practice and avoids clashes. Now watch the following:

```
local n = 1

function sum(a,b)
  n = n + 1
  return a + b
end

function report()
  print("number of summations: " .. n)
end
```

Here the variable `n` is visible after its definition and accessible for the two global functions. Actually the variable is visible to all the code following, unless of course we define a new variable with the same name. We can hide `n` as follows:

```
do
  local n = 1

  sum = function(a,b)
    n = n + 1
    return a + b
  end

  report = function()
    print("number of summations: " .. n)
  end
end
```

This example also shows another way of defining the function: by assignment.

The `do ... end` creates a so called closure. There are many places where such closures are created, for instance in function bodies or branches like `if ... then ... else`. This means that in the following snippet, variable `b` is not seen after the end:

```
if a > 10 then
```

```
  local b = a + 10
  print(b*b)
end
```

When you process a blob of LUA code in TEX (using `\directlua` or `\latelua`) it happens in a closure with an implied `do ... end`. So, `local` defined variables are really local.

TEX's data types

We mentioned `numbers`. At the TEX end we have counters as well as dimensions. Both are numbers but dimensions are specified differently

```
local n = tex.count[0]
local m = tex.dimen.lineheight
local o = tex.sp("10.3pt") -- scaled point is the smallest unit
```

The unit of dimension is 'scaled point' and this is a pretty small unit: 10 points equals to 655360 such units.

Another accessible data type is tokens. They are automatically converted to strings and vice versa.

```
tex.toks[0] = "message"
print(tex.toks[0])
```

Be aware of the fact that the tokens are letters so the following will come out as text and not issue a message:

```
tex.toks[0] = "\message{just text}"
print(tex.toks[0])
```

Control structures

Loops are not much different from other languages: we have `for ... do`, `while ... do` and `repeat ... until`. We start with the simplest case:

```
for index=1,10 do
  print(index)
end
```

You can specify a step and go downward as well:

```
for index=22,2,-2 do
  print(index)
end
```

Indexed tables can be traversed this way:

```
for index=1,#list do
  print(index, list[index])
end
```

Hashed tables on the other hand are dealt with as follows:

```
for key, value in next, list do
  print(key, value)
end
```

Here `next` is a built in function. There is more to say about this mechanism but the average user will use only this variant. Slightly less efficient is the following, more readable variant:

```
for key, value in pairs(list) do
  print(key, value)
end
```

and for an indexed table:

```
for index, value in ipairs(list) do
  print(index, value)
end
```

The function call to `pairs(list)` returns `next, list` so there is an (often neglectable) extra overhead of one function call.

The other two loop variants, `while` and `repeat`, are similar.

```
i = 0
while i < 10  do
  i = i + 1
  print(i)
end
```

This can also be written as:

```
i = 0
repeat
  i = i + 1
  print(i)
until i == 10
```

or:

```
i = 0
while true do
  i = i + 1
  print(i)
  if i == 10 then
    break
  end
end
```

Of course you can use more complex expressions in such constructs.

Conditions

Conditions have the following form:

```
if a == b or c > d or e then
  ...
elseif f == g then
  ...
else
  ...
end
```

Watch the double `==`. The complement of this is `~=`. Precedence is similar to other languages. In practice, as strings are hashed, tests like

```
if key == "first" then
  ...
end
```

and

```
if n == 1 then
  ...
end
```

are equally efficient. There is really no need to use numbers to identify states instead of more verbose strings.

Namespaces

Functionality can be grouped in libraries. There are a few default libraries, like `string`, `table`, `lpeg`, `math`, `io` and `os` and LuaTeX adds some more, like `node`, `tex` and `texio`.

A library is in fact nothing more than a bunch of functionality organized using a table, where the table provides a namespace as well as place to store public variables. Of course there can be local (hidden) variables used in defining functions.

```
do
  mylib = { }

  local n = 1

  function mylib.sum(a,b)
    n = n + 1
    return a + b
  end

  function mylib.report()
```

```
    print("number of summations: " .. n)
  end
end
```

The defined function can be called like:

```
mylib.report()
```

You can also create a shortcut, This speeds up the process because there are less lookups then. In the following code multiple calls take place:

```
local sum = mylib.sum

for i=1,10 do
  for j=1,10 do
    print(i, j, sum(i,j))
  end
end

mylib.report()
```

As LUA is pretty fast you should not overestimate the speedup, especially not when a function is called seldom. There is an important side effect here: in the case of:

```
print(i, j, sum(i,j))
```

the meaning of `sum` is frozen. But in the case of

```
print(i, j, mylib.sum(i,j))
```

the current meaning is taken, that is: each time the interpreter will access `mylib` and get the current meaning of `sum`. And there can be a good reason for this, for instance when the meaning is adapted to different situations.

In CONTEXT we have quite some code organized this way. Although much is exposed (if only because it is used all over the place) you should be careful in using functions (and data) that are still experimental. There are a couple of general libraries and some extend the core LUA libraries. You might want to take a look at the files in the distribution that start with `l-`, like `l-table.lua`. These files are preloaded.[2] For instance, if you want to inspect a table, you can say:

```
local t = { "aap", "noot", "mies" }
table.print(t)
```

You can get an overview of what is implemented by running the following command:

```
context s-tra-02 --mode=tablet
```

[2] In fact, if you write scripts that need their functionality, you can use `mtxrun` to process the script, as `mtxrun` has the core libraries preloaded as well.

Comment

You can add comments to your Lua code. There are basically two methods: one liners and multi line comments.

```
local option = "test" -- use this option with care

local method = "unknown" --[[comments can be very long and when
                   entered this way they can span multiple lines]]
```

The so called long comments look like long strings preceded by `--` and there can be more complex boundary sequences.

Pitfalls

Sometimes `nil` can bite you, especially in tables, as they have a dual nature: indexed as well as hashed.

```
\startluacode
local n1 = # { nil, 1, 2, nil }      -- 3
local n2 = # { nil, nil, 1, 2, nil } -- 0

context("n1 = %s and n2 = %s",n1,n2)
\stopluacode
```

results in: n1 = 3 and n2 = 0. So, you cannot really depend on the length operator here. On the other hand, with:

```
\startluacode
local function check(...)
    return select("#",...)
end

local n1 = check ( nil, 1, 2, nil )      -- 4
local n2 = check ( nil, nil, 1, 2, nil ) -- 5

context("n1 = %s and n2 = %s",n1,n2)
\stopluacode
```

we get: n1 = 4 and n2 = 5, so the `select` is quite useable. However, that function also has its specialities. The following example needs some close reading:

```
\startluacode
local function filter(n,...)
    return select(n,...)
end

local v1 = { filter ( 1, 1, 2, 3 ) }
local v2 = { filter ( 2, 1, 2, 3 ) }
```

```
local v3 = { filter ( 3, 1, 2, 3 ) }

context("v1 = %+t and v2 = %+t and v3 = %+t",v1,v2,v3)
\stopluacode
```

We collect the result in a table and show the concatenation: v1 = 1+2+3 and v2 = 2+3 and v3 = 3. So, what you effectively get is the whole list starting with the given offset.

```
\startluacode
local function filter(n,...)
    return (select(n,...))
end

local v1 = { filter ( 1, 1, 2, 3 ) }
local v2 = { filter ( 2, 1, 2, 3 ) }
local v3 = { filter ( 3, 1, 2, 3 ) }

context("v1 = %+t and v2 = %+t and v3 = %+t",v1,v2,v3)
\stopluacode
```

Now we get: v1 = 1 and v2 = 2 and v3 = 3. The extra `()` around the result makes sure that we only get one return value.

Of course the same effect can be achieved as follows:

```
local function filter(n,...)
    return select(n,...)
end

local v1 = filter ( 1, 1, 2, 3 )
local v2 = filter ( 2, 1, 2, 3 )
local v3 = filter ( 3, 1, 2, 3 )

context("v1 = %s and v2 = %s and v3 = %s",v1,v2,v3)
```

A few suggestions

You can wrap all kind of functionality in functions but sometimes it makes no sense to add the overhead of a call as the same can be done with hardly any code.

If you want a slice of a table, you can copy the range needed to a new table. A simple version with no bounds checking is:

```
local new = { } for i=a,b do new[#new+1] = old[i] end
```

Another, much faster, variant is the following.

```
local new = { unpack(old,a,b) }
```

You can use this variant for slices that are not extremely large. The function `table.sub` is an equivalent:

```
local new = table.sub(old,a,b)
```

An indexed table is empty when its size equals zero:

```
if #indexed == 0 then ... else ... end
```

Sometimes this is better:

```
if indexed and #indexed == 0 then ... else ... end
```

So how do we test if a hashed table is empty? We can use the `next` function as in:

```
if hashed and next(indexed) then ... else ... end
```

Say that we have the following table:

```
local t = { a=1, b=2, c=3 }
```

The call `next(t)` returns the first key and value:

```
local k, v = next(t)   -- "a", 1
```

The second argument to `next` can be a key in which case the following key and value in the hash table is returned. The result is not predictable as a hash is unordered. The generic for loop uses this to loop over a hashed table:

```
for k, v in next, t do
    ...
end
```

Anyway, when `next(t)` returns zero you can be sure that the table is empty. This is how you can test for exactly one entry:

```
if t and not next(t,next(t)) then ... else ... end
```

Here it starts making sense to wrap it into a function.

```
function table.has_one_entry(t)
    t and not next(t,next(t))
end
```

On the other hand, this is not that usefull, unless you can spend the runtime on it:

```
function table.is_empty(t)
    return not t or not next(t)
end
```

Interfacing

We have already seen that you can embed Lua code using commands like:

```
\startluacode
    print("this works")
\stopluacode
```

This command should not be confused with:

```
\startlua
    print("this works")
\stoplua
```

The first variant has its own catcode regime which means that tokens between the start and stop command are treated as LUA tokens, with the exception of TEX commands. The second variant operates under the regular TEX catcode regime.

Their short variants are `\ctxluacode` and `\ctxlua` as in:

```
\ctxluacode{print("this works")}
\ctxlua{print("this works")}
```

In practice you will probably use `\startluacode` when using or defining a blob of LUA and `\ctxlua` for inline code. Keep in mind that the longer versions need more initialization and have more overhead.

There are some more commands. For instance `\ctxcommand` can be used as an efficient way to access functions in the `commands` namespace. The following two calls are equivalent:

```
\ctxlua    {commands.thisorthat("...")}
\ctxcommand         {thisorthat("...")}
```

There are a few shortcuts to the `context` namespace. Their use can best be seen from their meaning:

```
\cldprocessfile#1{\directlua{context.runfile("#1")}}
\cldloadfile   #1{\directlua{context.loadfile("#1")}}
\cldcontext    #1{\directlua{context(#1)}}
\cldcommand    #1{\directlua{context.#1}}
```

Each time a call out to LUA happens the argument eventually gets parsed, converted into tokens, then back into a string, compiled to bytecode and executed. The next example code shows a mechanism that avoids this:

```
\startctxfunction MyFunctionA
    context(" A1 ")
\stopctxfunction

\startctxfunctiondefinition MyFunctionB
    context(" B2 ")
\stopctxfunctiondefinition
```

The first command associates a name with some LUA code and that code can be executed using:

```
\ctxfunction{MyFunctionA}
```

The second definition creates a command, so there we do:

```
\MyFunctionB
```

There are some more helpers but for use in document sources they make less sense. You can always browse the source code for examples.

Getting started

Some basics

I assume that you have either the so called ConTeXt standalone (formerly known as minimals) installed or TeXLive. You only need LuaTeX and can forget about installing pdfTeX or XeTeX, which saves you some megabytes and hassle. Now, from the users perspective a ConTeXt run goes like:

```
context yourfile
```

and by default a file with suffix `tex`, `mkiv`, or `mkvi` 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
context yourfile.mp
context yourfile.xyz --forcemp
```

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 interface is not that large. If you know ConTeXt, and if you know how to call commands, you basically can use this Lua method.

The examples that I will give are either (sort of) standalone, i.e. they are dealt with from Lua, or they are run within this document. Therefore you will see two patterns. If you want to make your own documentation, then you can use this variant:

```
\startbuffer
context("See this!")
\stopbuffer
```

```
\typebuffer \ctxluabuffer
```

I use anonymous buffers here but you can also use named ones. The other variant is:

```
\startluacode
context("See this!")
\stopluacode
```

This will process the code directly. Of course we could have encoded this document completely in LUA but that is not much fun for a manual.

The main command

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 matters:

`nothing`	:	just the command, no arguments
`string`	:	an argument with curly braces
`array`	:	a list between square brackets (sometimes optional)
`hash`	:	an assignment list between square brackets
`boolean`	:	when `true` a newline is inserted
	:	when `false`, omit braces for the next argument

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 blob of code is equivalent to:

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

You can simplify the third line of the LUA code to:

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

In case you wonder what the distinction is between square brackets and curly braces: the first category of arguments concerns settings or lists of options or names of instances while the second category normally concerns some text to be typeset.

Strings are interpreted as TEX input, so:

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

and if you don't want to escape:

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

are both correct. 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.

Spaces and Lines

In a TEX file, spaces and newline characters are collapsed into one space. The same happens in LUA. Compare the following examples. First we omit spaces:

```
context("left")
context("middle")
context("right")
```

resulting in: leftmiddleright. Next we add spaces:

```
context("left")
context(" middle ")
context("right")
```

resulting in: left middle right. We can also add more spaces:

```
context("left ")
context(" middle ")
context(" right")
```

resulting in: left middle right. In principle all content becomes a stream and after that the TEX parser will do its normal work: collapse spaces unless configured to do otherwise.

Now take the following code:

```
context("before")
context("word 1")
context("word 2")
context("word 3")
context("after")
```

resulting in: beforeword 1word 2word 3after. Here we get no spaces between the words at all, which is what we expect. So, how do we get lines (or paragraphs)?

```
context("before")
context.startlines()
context("line 1")
context("line 2")
context("line 3")
context.stoplines()
context("after")
```

results in: before

line 1line 2line 3

after.

This does not work out well, as again there are no lines seen at the TEX end. Newline tokens are injected by passing `true` to the `context` command:

```
context("before")
context.startlines()
context("line 1") context(true)
context("line 2") context(true)
context("line 3") context(true)
context.stoplines()
context("after")
```

resulting in: before

line 1
line 2
line 3

after. Don't confuse this with:

```
context("before") context.par()
context("line 1") context.par()
context("line 2") context.par()
context("line 3") context.par()
context("after")  context.par()
```

which results in: before

line 1

line 2

line 3

after. There we use the regular `\par` command to finish the current paragraph and normally you will use that method. In that case, when set, whitespace will be added between paragraphs.

This newline issue is a somewhat unfortunate inheritance of traditional TEX, where `\n` and `\r` mean something different. I'm still not sure if the CLD do the right thing as dealing with these tokens also depends on the intended effect. Catcodes as well as the LUATEX input parser also play a role. Anyway, the following also works:

```
context.startlines()
context("line 1\n")
context("line 2\n")
context("line 3\n")
context.stoplines()
```

Direct output

The CONTEXT user interface is rather consistent and the use of special input syntaxes is discouraged. Therefore, the LUA interface using tables and strings works quite well. However, imagine that you need to support some weird macro (or a primitive) that does not expect its argument between curly braces or brackets. The way out is to precede an argument by another one with the value `false`. We call this the direct interface. This is demonstrated in the following example.

```
\unexpanded\def\bla#1{[#1]}
\startluacode
context.bla(false,"***")
context.par()
context.bla("***")
\stopluacode
```

This results in: [*]**

[***]. Here, the first call results in three `*` being passed, and `#1` picks up the first token. The second call to `bla` gets `{***}` passed so here `#1` gets the triplet. In practice you will seldom need the direct interface.

In ConTeXt for historical reasons, combinations accept the following syntax:

```
\startcombination % optional specification, like [2*3]
  {\framed{content one}} {caption one}
  {\framed{content two}} {caption two}
\stopcombination
```

You can also say:

```
\startcombination
  \combination {\framed{content one}} {caption one}
  \combination {\framed{content two}} {caption two}
\stopcombination
```

When coded in Lua, we can feed the first variant as follows:

```
context.startcombination()
  context.direct("one","two")
  context.direct("one","two")
context.stopcombination()
```

To give you an idea what this looks like, we render it:

one one
two two

So, the `direct` function is basically a no-op and results in nothing by itself. Only arguments are passed. An equivalent but bit more ugly looking is:

```
context.startcombination()
  context(false,"one","two")
  context(false,"one","two")
context.stopcombination()
```

Catcodes

If you are familiar with the inner working of TEX, you will know that characters can have special meanings. This meaning is determined by their catcodes.

```
context("$x=1$")
```

This gives: $x = 1$ because the dollar tokens trigger inline math mode. If you think that this is annoying, you can do the following:

```
context.pushcatcodes("text")
context("$x=1$")
context.popcatcodes()
```

Now we get: $x=1$. There are several catcode regimes of which only a few make sense in the perspective of the cld interface.

`ctx, ctxcatcodes, context`	the normal CONTEXT catcode regime
`prt, prtcatcodes, protect`	the CONTEXT protected regime, used for modules
`tex, texcatcodes, plain`	the traditional (plain) TEX regime
`txt, txtcatcodes, text`	the CONTEXT regime but with less special characters
`vrb, vrbcatcodes, verbatim`	a regime specially meant for verbatim
`xml, xmlcatcodes`	a regime specially meant for XML processing

In the second case you can still get math:

```
context.pushcatcodes("text")
context.mathematics("x=1")
context.popcatcodes()
```

When entering a lot of math you can also consider this:

```
context.startimath()
context("x")
context("=")
context("1")
context.stopimath()
```

Module writers can use `unprotect` and `protect` as they do at the TEX end.

As we've seen, a function call to `context` acts like a print, as in:

```
context("test ")
context.bold("me")
context(" first")
```

resulting in: test **me** first. When more than one argument is given, the first argument is considered a format conforming the `string.format` function:

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

resulting in: $\pi = 3.14159$. This means that when you say:

```
context(a,b,c,d,e,f)
```

the variables `b` till `f` are passed to the format and when the format does not use them, they will not end up in your output.

```
context("%s %s %s",1,2,3)
context(1,2,3)
```

The first line results in the three numbers being typeset, but in the second case only the number 1 is typeset.

More on functions

Why we need them

In a previous section we introduced functions as arguments. At first sight this feature looks strange but you need to keep in mind that a call to a `context` function has no direct consequences. It generates TEX code that is executed after the current LUA chunk ends and control is passed back to TEX. Take the following code:

```
context.framed( {
    frame = "on",
    offset = "5mm",
    align = "middle"
  },
  context.input("knuth")
)
```

We call the function `framed` but before the function body is executed, the arguments get evaluated. This means that `input` gets processed before `framed` gets done. As a result there is no second argument to `framed` and no content gets passed: an error is reported. This is why we need the indirect call:

```
context.framed( {
    frame = "on",
    align = "middle"
  },
  function() context.input("knuth") end
)
```

This way we get what we want:

Thus, I came to the conclusion that the designer of a new system must not only be the implementer and first large–scale user; the designer should also write the first user manual. The separation of any of these four components would have hurt TEX significantly. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important. But a system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments.

The function is delayed till the `framed` command is executed. If your applications use such calls a lot, you can of course encapsulate this ugliness:

```
mycommands = mycommands or { }

function mycommands.framed_input(filename)
  context.framed( {
    frame = "on",
    align = "middle"
  },
  function() context.input(filename) end
end

mycommands.framed_input("knuth")
```

Of course you can nest function calls:

```
context.placefigure(
  "caption",
  function()
    context.framed( {
      frame = "on",
      align = "middle"
    },
      function() context.input("knuth") end
    )
  end
)
```

Or you can use a more indirect method:

```
function text()
  context.framed( {
      frame = "on",
      align = "middle"
```

```
    },
    function() context.input("knuth") end
  )
end

context.placefigure(
  "none",
  function() text() end
)
```

You can develop your own style and libraries just like you do with regular Lua code. Browsing the already written code can give you some ideas.

How we can avoid them

As many nested functions can obscure the code rather quickly, there is an alternative. In the following examples we use `test`:

```
\def\test#1{[#1]}
```

```
context.test("test 1 ",context("test 2a")," test 3")
```

This gives: test 2a[test 1] test 3. As you can see, the second argument is executed before the encapsulating call to `test`. So, we should have packed it into a function but here is an alternative:

```
context.test("test 1 ",context.delayed("test 2a")," test 3")
```

Now we get: [test 1]test 2a test 3. We can also delay functions themselves, look at this:

```
context.test("test 1 ",context.delayed.test("test 2b")," test 3")
```

The result is: [test 1][test 2b] test 3. This feature also conveniently permits the use of temporary variables, as in:

```
local f = context.delayed.test("test 2c")
context("before ",f," after")
```

Of course you can limit the amount of keystrokes even more by creating a shortcut:

```
local delayed = context.delayed

context.test("test 1 ",delayed.test("test 2")," test 3")
context.test("test 4 ",delayed.test("test 5")," test 6")
```

So, if you want you can produce rather readable code and readability of code is one of the reasons why Lua was chosen in the first place. This is a good example of why coding in TeX makes sense as it looks more intuitive:

```
\test{test 1 \test{test 2} test 3}
\test{test 4 \test{test 5} test 6}
```

There is also another mechanism available. In the next example the second argument is actually a string.

```
local nested = context.nested

context.test("test 8",nested.test("test 9"),"test 10")
```

There is a pitfall here: a nested context command needs to be flushed explicitly, so in the case of:

```
context.nested.test("test 9")
```

a string is created but nothing ends up at the TEX end. Flushing is up to you. Beware: `nested` only works with the regular CONTEXT catcode regime.

Trial typesetting

Some typesetting mechanisms demand a preroll. For instance, when determining the most optimal way to analyse and therefore typeset a table, it is necessary to typeset the content of cells first. Inside CONTEXT there is a state tagged 'trial typesetting' which signals other mechanisms that for instance counters should not be incremented more than once.

Normally you don't need to worry about these issues, but when writing the code that implements the LUA interface to CONTEXT, it definitely had to be taken into account as we either or not can free cached (nested) functions.

You can influence this caching to some extend. If you say

```
function()
  context("whatever")
end
```

the function will be removed from the cache when CONTEXT is not in the trial typesetting state. You can prevent removal of a function by returning `true`, as in:

```
function()
  context("whatever")
  return true
end
```

Whenever you run into a situation that you don't get the outcome that you expect, you can consider returning `true`. However, keep in mind that it will take more memory, something that only matters on big runs. You can force flushing the whole cache by:

```
context.restart()
```

An example of an occasion where you need to keep the function available is in repeated content, for instance in headers and footers.

```
context.setupheadertexts {
  function()
```

```
    context.pagenumber()
    return true
  end
}
```

Of course it is not needed when you use the following method:

```
context.pagenumber("pagenumber")
```

Because here ConTEXt itself deals with the content driven by the keyword `pagenumber`.

Steppers

The `context` commands are accumulated within a `\ctxlua` call and only after the call is finished, control is back at the TEX end. Sometimes you want (in your Lua code) to go on and pretend that you jump out to TEX for a moment, but come back to where you left. The stepper mechanism permits this.

A not so practical but nevertheless illustrative example is the following:

```
\startluacode
  context.stepwise (function()
    context.startitemize()
       context.startitem()
         context.step("BEFORE 1")
       context.stopitem()
       context.step("\\setbox0\\hbox{!!!!}")
       context.startitem()
         context.step("%p",tex.getbox(0).width)
       context.stopitem()
       context.startitem()
         context.step("BEFORE 2")
       context.stopitem()
       context.step("\\setbox2\\hbox{????}")
       context.startitem()
         context.step("%p",tex.getbox(2).width)
       context.startitem()
         context.step("BEFORE 3")
       context.stopitem()
       context.startitem()
         context.step("\\copy0\\copy2")
       context.stopitem()
       context.startitem()
         context.step("BEFORE 4")
         context.startitemize()
           context.stepwise (function()
```

```
            context.step("\\bgroup")
            context.step("\\setbox0\\hbox{>>>>}")
            context.startitem()
              context.step("%p",tex.getbox(0).width)
            context.stopitem()
            context.step("\\setbox2\\hbox{<<<<}")
            context.startitem()
              context.step("%p",tex.getbox(2).width)
            context.stopitem()
            context.startitem()
              context.step("\\copy0\\copy2")
            context.stopitem()
            context.startitem()
              context.step("\\copy0\\copy2")
            context.stopitem()
            context.step("\\egroup")
          end)
        context.stopitemize()
      context.stopitem()
      context.startitem()
        context.step("AFTER 1\\par")
      context.stopitem()
      context.startitem()
        context.step("\\copy0\\copy2\\par")
      context.stopitem()
      context.startitem()
        context.step("\\copy0\\copy2\\par")
      context.stopitem()
      context.startitem()
        context.step("AFTER 2\\par")
      context.stopitem()
      context.startitem()
        context.step("\\copy0\\copy2\\par")
      context.stopitem()
      context.startitem()
        context.step("\\copy0\\copy2\\par")
      context.stopitem()
    context.stopitemize()
end)
\stopluacode
```

This gives an (ugly) itemize with a nested one:

- BEFORE 1
- 12.76001pt
- BEFORE 2
- 18.88pt
- BEFORE 3
- !!!!????
- BEFORE 4
  - 31.12pt
  - 31.12pt
  - >>>><<<<
  - >>>><<<<
- AFTER 1
- !!!!????
- !!!!????
- AFTER 2
- !!!!????
- !!!!????

As you can see in the code, the `step` call accepts multiple arguments, but when more than one argument is given the first one is treated as a formatter.

A few Details

Variables

Normally it makes most sense to use the English version of ConTeXt. The advantage is that you can use English keywords, as in:

```
context.framed( {
    frame = "on",
```

```
  },
  "some text"
)
```

If you use the Dutch interface it looks like this:

```
context.omlijnd( {
    kader = "aan",
  },
  "wat tekst"
)
```

A rather neutral way is:

```
context.framed( {
    frame = interfaces.variables.on,
  },
  "some text"
)
```

But as said, normally you will use the English user interface so you can forget about these matters. However, in the ConTEXt core code you will often see the variables being used this way because there we need to support all user interfaces.

Modes

ConTEXt carries a concept of modes. You can use modes to create conditional sections in your style (and/or content). You can control modes in your styles or you can set them at the command line or in job control files. When a mode test has to be done at processing time, then you need constructs like the following:

```
context.doifmodeelse( "screen",
  function()
      ... -- mode == screen
  end,
  function()
      ... -- mode ~= screen
  end
)
```

However, often a mode does not change during a run, and then we can use the following method:

```
if tex.modes["screen"] then
  ...
else
  ...
end
```

Watch how the `modes` table lives in the `tex` namespace. We also have `systemmodes`. At the TEX end these are mode names preceded by a `*`, so the following code is similar:

```
if tex.modes["*mymode"] then
  -- this is the same
elseif tex.systemmodes["mymode"] then
  -- test as this
else
  -- but not this
end
```

Inside CONTEXT we also have so called constants, and again these can be consulted at the LUA end:

```
if tex.constants["someconstant"] then
  ...
else
  ...
end
```

But you will hardly need these and, as they are often not public, their meaning can change, unless of course they *are* documented as public.

Token lists

There is normally no need to mess around with nodes and tokens at the LUA end yourself. However, if you do, then you might want to flush them as well. Say that at the TEX end we have said:

```
\toks0 = {Don't get \inframed{framed}!}
```

Then at the LUA end you can say:

```
context(tex.toks[0])
```

and get: Don't get framed! In fact, token registers are exposed as strings so here, register zero has type `string` and is treated as such.

```
context("< %s >",tex.toks[0])
```

This gives: < Don't get framed! >. But beware, if you go the reverse way, you don't get what you might expect:

```
tex.toks[0] = [[\framed{oeps}]]
```

If we now say `\the\toks0` we will get \framed{oeps} as all tokens are considered to be letters.

Node lists

If you're not deep into TEX you will never feel the need to manipulate node lists yourself, but you might want to flush boxes. As an example we put something in box zero (one of the scratch boxes).

```
\setbox0 = \hbox{Don't get \inframed{framed}!}
```

At the TeX end you can flush this box (`\box0`) or take a copy (`\copy0`). At the Lua end you would do:

```
context.copy()
context.direct(0)
```

or:

```
context.copy(false,0)
```

but this works as well:

```
context(node.copy_list(tex.box[0]))
```

So we get: Don't get framed! If you do:

```
context(tex.box[0])
```

you also need to make sure that the box is freed but let's not go into those details now.

Here is an example of messing around with node lists that get seen before a paragraph gets broken into lines, i.e. when hyphenation, font manipulation etc. take place. First we define some colors:

```
\definecolor[mynesting:0][r=.6]
\definecolor[mynesting:1][g=.6]
\definecolor[mynesting:2][r=.6,g=.6]
```

Next we define a function that colors nodes in such a way that we can see the different processing stages.

```
\startluacode
local enabled  = false
local count    = 0
local setcolor = nodes.tracers.colors.set

function userdata.processmystuff(head)
    if enabled then
        local color = "mynesting:" .. (count % 3)
     -- for n in node.traverse(head) do
        for n in node.traverse_id(nodes.nodecodes.glyph,head) do
            setcolor(n,color)
        end
        count = count + 1
        return head, true
    end
    return head, false
end
```

```
function userdata.enablemystuff()
    enabled = true
end

function userdata.disablemystuff()
    enabled = false
end
\stopluacode
```

We hook this function into the normalizers category of the processor callbacks:

```
\startluacode
nodes.tasks.appendaction(
  "processors",
  "normalizers",
  "userdata.processmystuff"
  )
\stopluacode
```

We now can enable this mechanism and show an example:

```
\startbuffer
Node lists are processed \hbox {nested from \hbox{inside} out} which
is not what you might expect. But, \hbox{coloring} does not \hbox
{happen} really nested here, more \hbox {in} \hbox {the} \hbox {order}
\hbox {of} \hbox {processing}.
\stopbuffer

\ctxlua{userdata.enablemystuff()}
\par \getbuffer \par
\ctxlua{userdata.disablemystuff()}
```

The `\par` is needed because otherwise the processing is already disabled before the paragraph gets seen by TEX. This is the result:

Node lists are processed nested from inside out which is not what you might expect. But, coloring does not happen really nested here, more in the order of processing.

Instead of using a boolean to control the state, we can also do this:

```
\startluacode
local count    = 0
local setcolor = nodes.tracers.colors.set

function userdata.processmystuff(head)
    count = count + 1
    local color = "mynesting:" .. (count % 3)
```

```
    for n in node.traverse_id(nodes.nodecodes.glyph,head) do
        setcolor(n,color)
    end
    return head, true
end

nodes.tasks.appendaction(
  "processors",
  "after",
  "userdata.processmystuff"
  )
\stopluacode
```

Disabling now happens with:

```
\startluacode
nodes.tasks.disableaction("processors", "userdata.processmystuff")
\stopluacode
```

As you might want to control these things in more details, a simple helper mechanism was made: markers. The following example code shows the way:

```
\definemarker[mymarker]
```

Again we define some colors:

```
\definecolor[mymarker:1][r=.6]
\definecolor[mymarker:2][g=.6]
\definecolor[mymarker:3][r=.6,g=.6]
```

The Lua code looks similar to the code presented before:

```
\startluacode
local setcolor    = nodes.tracers.colors.setlist
local getmarker   = nodes.markers.get
local hlist_code  = nodes.codes.hlist
local traverse_id = node.traverse_id

function userdata.processmystuff(head)
    for n in traverse_id(hlist_code,head) do
        local m = getmarker(n,"mymarker")
        if m then
            setcolor(n.list,"mymarker:" .. m)
        end
    end
    return head, true
end
```

```
nodes.tasks.appendaction(
  "processors",
  "after",
  "userdata.processmystuff")
nodes.tasks.disableaction("processors", "userdata.processmystuff")
\stopluacode
```

This time we disabled the processor (if only because in this document we don't want the overhead).

```
\startluacode
nodes.tasks.enableaction("processors", "userdata.processmystuff")
\stopluacode
 Node lists are processed \hbox \boxmarker{mymarker}{1}
{nested from \hbox{inside} out}
which is not what you might expect. But,
\hbox {coloring} does not \hbox {happen} really
nested here, more \hbox {in} \hbox \boxmarker{mymarker}{2}
{the} \hbox {order} \hbox {of} \hbox \boxmarker{mymarker}{3}
{processing}.

\startluacode
nodes.tasks.disableaction("processors", "userdata.processmystuff")
\stopluacode
```

The result looks familiar:

Node lists are processed nested from inside out which is not what you might expect. But, coloring does not happen really nested here, more in the order of processing.

Some more examples

Appetizer

Before we give some more examples, we will have a look at the way the title page is made. This way you get an idea what more is coming.

```
local todimen, random = number.todimen, math.random

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",
    },
    ""
)
local settings = {
    frame = "off",
    background = "color",
    backgroundcolor = secondcolor,
    foregroundcolor = firstcolor,
    foregroundstyle = "type",
}
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 = random(360),
            },
            settings,
            "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 feel a bit more natural in LUA then in TEX. The result is shown in figure 1. The actual cover page of the manual was derived from this.

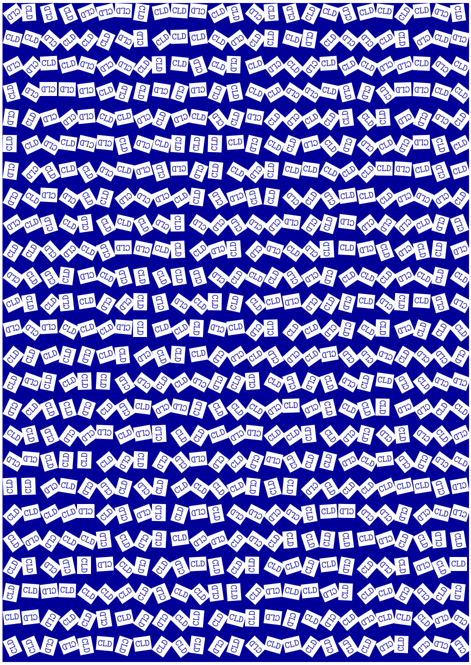


Figure 1 The simplified cover page.

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

resulting in:

a. this is item 1
b. this is item 2
c. this is item 3
d. this is item 4
e. this is item 5
f. this is item 6
g. this is item 7
h. this is item 8
i. this is item 9
j. this is item 10

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:

```
local one = {
  align = "middle",
  style = "type",
}
local two = {
  align = "middle",
  style = "type",
  background = "color",
  backgroundcolor = "darkblue",
  foregroundcolor = "white",
}
local random = math.random
context.bTABLE { framecolor = "darkblue" }
    for i=1,10 do
      context.bTR()
      for i=1,20 do
          local r = random(99)
          context.bTD(r < 50 and one or two)
          context("%2i",r)
          context.eTD()
      end
      context.eTR()
    end
context.eTABLE()
```

Here we see a function call to `context` in the most indented line. The first argument is a format that makes sure that we get two digits and the random number is substituted into this format. The result is shown in table 1. The

28	95	39	94	37	52	38	66	98	46	30	90	39	77	62	41	49	77	80	68
22	11	17	97	67	60	7	88	6	71	48	33	67	87	28	4	40	65	70	38
12	99	29	51	77	90	91	27	67	72	94	88	82	12	86	49	72	93	38	77
65	85	11	32	73	38	36	13	4	6	50	15	6	79	66	82	69	57	9	37
30	4	26	12	16	13	61	87	6	98	65	70	83	76	2	57	14	38	69	18
43	20	33	48	98	98	31	68	56	40	5	85	43	31	97	59	43	58	46	49
57	12	19	41	87	21	97	2	58	66	19	2	85	52	50	83	50	80	51	6
21	56	91	64	87	88	23	31	47	69	79	4	80	98	44	68	19	41	70	77
8	89	78	92	41	29	76	90	9	28	96	30	84	87	93	71	76	16	2	23
84	80	26	65	79	70	34	98	12	4	75	19	92	54	12	33	82	87	24	91

Table 1 A table generated by Lua.

line correction is ignored when we use this table as a float, otherwise it assures proper vertical spacing around the table. Watch how we define the tables `one` and `two` beforehand. This saves 198 redundant table constructions.

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

```
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 (we already showed some alternative approaches in previous chapters). A function argument is treated as special and in this case the external figure ends up right. Here is another example:

```
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)
```

In this case the figure is not an argument so it gets flushed sequentially with the rest:

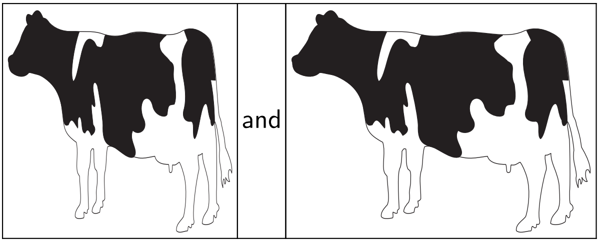


Figure 2 Two cows!

Styles

Say that you want to typeset a word in a bold font. You can do that this way:

```
context("This is ")
context.bold("important")
context("!")
```

Now imagine that you want this important word to be in red too. As we have a nested command, we end up with a nested call:

```
context("This is ")
context.bold(function() context.color( { "red" }, "important") end)
context("!")
```

or

```
context("This is ")
context.bold(context.delayed.color( { "red" }, "important"))
context("!")
```

In that case it's good to know that there is a command that combines both features:

```
context("This is ")
context.style( { style = "bold", color = "red" }, "important")
context("!")
```

But that is still not convenient when we have to do that often. So, you can wrap the style switch in a function.

```
local function mycommands.important(str)
    context.style( { style = "bold", color = "red" }, str )
end
context("This is ")
mycommands.important( "important")
context(", and ")
mycommands.important( "this")
context(" too !")
```

Or you can setup a named style:

```
context.setupstyle( { "important" },
                    { style = "bold", color = "red" } )
context("This is ")
context.style( { "important" }, "important")
context(", and ")
context.style( { "important" }, "this")
context(" too !")
```

Or even define one:

```
context.definestyle( { "important" },
                     { style = "bold", color = "red" } )
context("This is ")
context.important("important")
context(", and ")
context.important("this")
context(" too !")
```

This last solution is especially handy for more complex cases:

```
context.definestyle( { "important" },
                     { style = "bold", color = "red" } )
context("This is ")
context.startimportant()
context.inframed("important")
context.stopimportant()
context(", and ")
context.important("this")
context(" too !")
```

resulting in: This is **important**, and **this** too !

A complete example

One day my 6 year old niece Lorien was at the office and wanted to know what I was doing. As I knew she was practicing arithmetic at school I wrote a quick and dirty script to generate sheets with exercises. The most impressive part was that the answers were included. It was a rather braindead bit of LUA, written in a few minutes, but the weeks after I ended up running it a few more times, for her and her friends, every time a bit more difficult and also using different arithmetic. It was that script that made me decide to extend the basic cld manual into this more extensive document.

We generate three columns of exercises. Each exercise is a row in a table. The last argument to the function determines if answers are shown.

```
local random = math.random

local function ForLorien(n,maxa,maxb,answers)
  context.startcolumns { n = 3 }
  context.starttabulate { "|r|c|r|c|r|" }
  for i=1,n do
    local sign = random(0,1) > 0.5
    local a, b = random(1,maxa or 99), random(1,max or maxb or 99)
    if b > a and not sign then a, b = b, a end
    context.NC()
    context(a)
    context.NC()
    context.mathematics(sign and "+" or "-")
    context.NC()
    context(b)
    context.NC()
    context("=")
    context.NC()
    context(answers and (sign and a+b or a-b))
    context.NC()
    context.NR()
  end
  context.stoptabulate()
  context.stopcolumns()
  context.page()
end
```

This is a typical example of where it's more convenient to write the code in LUA that in TEX's macro language. As a consequence setting up the page also happens in LUA:

```
context.setupbodyfont {
  "palatino",
  "14pt"
}
```

```
context.setuplayout {
  backspace = "2cm",
  topspace  = "2cm",
  header    = "1cm",
  footer    = "0cm",
  height    = "middle",
  width     = "middle",
}
```

This leaves us to generate the document. There is a pitfall here: we need to use the same random number for the exercises and the answers, so we freeze and defrost it. Functions in the `commands` namespace implement functionality that is used at the TEX end but better can be done in LUA than in TEX macro code. Of course these functions can also be used at the LUA end.

```
context.starttext()

  local n = 120

  commands.freezerandomseed()

  ForLorien(n,10,10)
  ForLorien(n,20,20)
  ForLorien(n,30,30)
  ForLorien(n,40,40)
  ForLorien(n,50,50)

  commands.defrostrandomseed()

  ForLorien(n,10,10,true)
  ForLorien(n,20,20,true)
  ForLorien(n,30,30,true)
  ForLorien(n,40,40,true)
  ForLorien(n,50,50,true)

context.stoptext()
```

A few pages of the result are shown in figure 3. In the CONTEXT distribution a more advanced version can be found in `s-edu-01.cld` as I was also asked to generate multiplication and table exercises. In the process I had to make sure that there were no duplicates on a page as she complained that was not good. There a set of sheets is generated with:

```
moduledata.educational.arithematic.generate {
  name     = "Bram Otten",
  fontsize = "12pt",
```

1

6 − 4 =	6 + 8 =	4 − 3 =
1 + 8 =	8 + 9 =	3 − 3 =
4 + 1 =	1 + 10 =	8 + 2 =
10 + 2 =	7 − 7 =	8 + 7 =
5 + 3 =	5 + 5 =	10 − 5 =
9 − 7 =	4 − 2 =	9 − 6 =
6 + 10 =	9 − 3 =	5 + 1 =
4 + 9 =	7 − 6 =	2 − 1 =
9 + 10 =	8 + 3 =	1 + 2 =
7 − 4 =	2 − 1 =	4 + 1 =
10 + 2 =	6 − 5 =	5 + 4 =
9 − 3 =	5 + 6 =	8 − 8 =
7 − 1 =	7 − 2 =	4 + 10 =
8 − 4 =	7 + 6 =	7 + 5 =
4 − 3 =	4 + 10 =	10 − 5 =
9 − 5 =	10 − 7 =	8 − 1 =
5 + 6 =	4 − 3 =	5 + 9 =
6 + 1 =	7 − 2 =	3 − 2 =
5 − 4 =	4 − 3 =	5 + 1 =
4 − 1 =	1 + 7 =	5 + 9 =
1 + 3 =	4 + 3 =	9 − 9 =
3 + 5 =	9 − 2 =	6 − 1 =
5 + 5 =	8 − 6 =	9 + 2 =
7 − 2 =	8 + 2 =	10 − 9 =
10 − 9 =	5 + 8 =	10 − 1 =
5 − 1 =	7 − 4 =	7 − 5 =
8 + 8 =	5 + 3 =	7 + 10 =
5 + 1 =	6 + 7 =	7 − 1 =
10 − 9 =	9 + 9 =	3 − 2 =
7 − 4 =	5 − 2 =	7 + 5 =
6 − 3 =	6 − 5 =	8 − 8 =
2 + 1 =	5 − 1 =	6 + 6 =
8 + 7 =	2 − 1 =	9 − 6 =
5 + 7 =	7 + 4 =	9 − 3 =
9 − 4 =	9 + 4 =	6 + 4 =
10 − 9 =	3 − 2 =	9 − 2 =
6 + 2 =	1 + 9 =	8 − 7 =
7 − 1 =	8 − 7 =	8 − 1 =
6 + 5 =	5 − 1 =	7 − 5 =
10 + 2 =	5 − 1 =	5 + 10 =

exercises

6

6 − 4 = 2	6 + 8 = 14	4 − 3 = 1
1 + 8 = 9	8 + 9 = 17	3 − 3 = 0
4 + 1 = 5	1 + 10 = 11	8 + 2 = 10
10 + 2 = 12	7 − 7 = 0	8 + 7 = 15
5 + 3 = 8	5 + 5 = 10	10 − 5 = 5
9 − 7 = 2	4 − 2 = 2	9 − 6 = 3
6 + 10 = 16	9 − 3 = 6	5 + 1 = 6
4 + 9 = 13	7 − 6 = 1	2 − 1 = 1
9 + 10 = 19	8 + 3 = 11	1 + 2 = 3
7 − 4 = 3	2 − 1 = 1	4 + 1 = 5
10 + 2 = 12	6 − 5 = 1	5 + 4 = 9
9 − 3 = 6	5 + 6 = 11	8 − 8 = 0
7 − 1 = 6	7 − 2 = 5	4 + 10 = 14
8 − 4 = 4	7 + 6 = 13	7 + 5 = 12
4 − 3 = 1	4 + 10 = 14	10 − 5 = 5
9 − 5 = 4	10 − 7 = 3	8 − 1 = 7
5 + 6 = 11	4 − 3 = 1	5 + 9 = 14
6 + 1 = 7	7 − 2 = 5	3 − 2 = 1
5 − 4 = 1	4 − 3 = 1	5 + 1 = 6
4 − 1 = 3	1 + 7 = 8	5 + 9 = 14
1 + 3 = 4	4 + 3 = 7	9 − 9 = 0
3 + 5 = 8	9 − 2 = 7	6 − 1 = 5
5 + 5 = 10	8 − 6 = 2	9 + 2 = 11
7 − 2 = 5	8 + 2 = 10	10 − 9 = 1
10 − 9 = 1	5 + 8 = 13	10 − 1 = 9
5 − 1 = 4	7 − 4 = 3	7 − 5 = 2
8 + 8 = 16	5 + 3 = 8	7 + 10 = 17
5 + 1 = 6	6 + 7 = 13	7 − 1 = 6
10 − 9 = 1	9 + 9 = 18	3 − 2 = 1
7 − 4 = 3	5 − 2 = 3	7 + 5 = 12
6 − 3 = 3	6 − 5 = 1	8 − 8 = 0
2 + 1 = 3	5 − 1 = 4	6 + 6 = 12
8 + 7 = 15	2 − 1 = 1	9 − 6 = 3
5 + 7 = 12	7 + 4 = 11	9 − 3 = 6
9 − 4 = 5	9 + 4 = 13	6 + 4 = 10
10 − 9 = 1	3 − 2 = 1	9 − 2 = 7
6 + 2 = 8	1 + 9 = 10	8 − 7 = 1
7 − 1 = 6	8 − 7 = 1	8 − 1 = 7
6 + 5 = 11	5 − 1 = 4	7 − 5 = 2
10 + 2 = 12	5 − 1 = 4	5 + 10 = 15

answers

Figure 3 Lorien's challenge.

```
  columns  = 2,
  run      = {
    { method = "bin_add_and_subtract", maxa =   8, maxb =   8 },
    { method = "bin_add_and_subtract", maxa =  16, maxb =  16 },
    { method = "bin_add_and_subtract", maxa =  32, maxb =  32 },
    { method = "bin_add_and_subtract", maxa =  64, maxb =  64 },
    { method = "bin_add_and_subtract", maxa = 128, maxb = 128 },
  },
}
```

Interfacing

The fact that we can define functionality using Lua code does not mean that we should abandon the TeX interface. As an example of this we use a relatively simple module for typesetting morse code.[3] First we create a proper namespace:

```
moduledata.morse = moduledata.morse or { }
local morse      = moduledata.morse
```

[3] The real module is a bit larger and can format verbose morse.

We will use a few helpers and create shortcuts for them. The first helper loops over each UTF character in a string. The other two helpers map a character onto an uppercase (because morse only deals with uppercase) or onto an similar shaped character (because morse only has a limited character set).

```
local utfcharacters = string.utfcharacters
local ucchars, shchars = characters.ucchars, characters.shchars
```

The morse codes are stored in a table.

```
local codes = {
    ["A"] = "·-",       ["B"] = "-···",
    ["C"] = "-·-·",     ["D"] = "-··",
    ["E"] = "·",        ["F"] = "··-·",
    ["G"] = "--·",      ["H"] = "····",
    ["I"] = "··",       ["J"] = "·---",
    ["K"] = "-·-",      ["L"] = "·-··",
    ["M"] = "--",       ["N"] = "-·",
    ["O"] = "---",      ["P"] = "·--·",
    ["Q"] = "--·-",     ["R"] = "·-·",
    ["S"] = "···",      ["T"] = "-",
    ["U"] = "··-",      ["V"] = "···-",
    ["W"] = "·--",      ["X"] = "-··-",
    ["Y"] = "-·--",     ["Z"] = "--··",

    ["0"] = "-----",    ["1"] = "·----",
    ["2"] = "··---",    ["3"] = "···--",
    ["4"] = "····-",    ["5"] = "·····",
    ["6"] = "-····",    ["7"] = "--···",
    ["8"] = "---··",    ["9"] = "----·",
    ["."] = "·-·-·-",   [","] = "--··--",
    [":"] = "---···",   [";"] = "-·-·-",
    ["?"] = "··--··",   ["!"] = "-·-·--",
    ["-"] = "-····-",   ["/"] = "-··-· ",
    ["("] = "-·--·",    [")"] = "-·--·-",
    ["="] = "-···-",    ["@"] = "·--·-·",
    ["'"] = "·----·",   ['"'] = "·-··-·",

    ["À"] = "·--·-",    ["Å"] = "·--·-",
    ["Ä"] = "·-·-",     ["Æ"] = "·-·-",
    ["Ç"] = "-·-··",    ["É"] = "··-··",
    ["È"] = "·-··-",    ["Ñ"] = "--·--",
    ["Ö"] = "---·",     ["Ø"] = "---·",
    ["Ü"] = "··--",     ["ß"] = "··· ···",
}
```

```
morse.codes = codes
```

As you can see, there are a few non ASCII characters supported as well. There will never be full UNICODE support simply because morse is sort of obsolete. Also, in order to support UNICODE one could as well use the bits of UTF characters, although … memorizing the whole UNICODE table is not much fun.

We associate a metatable index function with this mapping. That way we can not only conveniently deal with the casing, but also provide a fallback based on the shape. Once found, we store the representation so that only one lookup is needed per character.

```
local function resolvemorse(t,k)
    if k then
        local u = ucchars[k]
        local v = rawget(t,u) or rawget(t,shchars[u]) or false
        t[k] = v
        return v
    else
        return false
    end
end

setmetatable(codes, { __index = resolvemorse })
```

Next comes some rendering code. As we can best do rendering at the TEX end we just use macros.

```
local MorseBetweenWords      = context.MorseBetweenWords
local MorseBetweenCharacters = context.MorseBetweenCharacters
local MorseLong              = context.MorseLong
local MorseShort             = context.MorseShort
local MorseSpace             = context.MorseSpace
local MorseUnknown           = context.MorseUnknown
```

The main function is not that complex. We need to keep track of spaces and newlines. We have a nested loop because a fallback to shape can result in multiple characters.

```
function morse.tomorse(str)
    local inmorse = false
    for s in utfcharacters(str) do
        local m = codes[s]
        if m then
            if inmorse then
                MorseBetweenWords()
            else
                inmorse = true
```

```
            end
            local done = false
            for m in utfcharacters(m) do
                if done then
                    MorseBetweenCharacters()
                else
                    done = true
                end
                if m == "·" then
                    MorseShort()
                elseif m == "-" then
                    MorseLong()
                elseif m == " " then
                    MorseBetweenCharacters()
                end
            end
            inmorse = true
        elseif s == "\n" or s == " " then
            MorseSpace()
            inmorse = false
        else
            if inmorse then
                MorseBetweenWords()
            else
                inmorse = true
            end
            MorseUnknown(s)
        end
    end
end
```

We use this function in two additional functions. One typesets a file, the other a table of available codes.

```
function morse.filetomorse(name,verbose)
    morse.tomorse(resolvers.loadtexfile(name),verbose)
end

function morse.showtable()
    context.starttabulate { "|l|l|" }
    for k, v in table.sortedpairs(codes) do
        context.NC() context(k)
        context.NC() morse.tomorse(v,true)
```

```
        context.NC() context.NR()
    end
    context.stoptabulate()
end
```

We're done with the LUA code that we can either put in an external file or put in the module file. The TEX file has two parts. The typesetting macros that we use at the LUA end are defined first. These can be overloaded.

```
\def\MorseShort
  {\dontleavehmode
   \vrule
      width  \MorseWidth
      height \MorseHeight
      depth  \zeropoint
   \relax}

\def\MorseLong
  {\dontleavehmode
   \vrule
      width 3\dimexpr\MorseWidth
      height         \MorseHeight
      depth          \zeropoint
   \relax}

\def\MorseBetweenCharacters
  {\kern\MorseWidth}

\def\MorseBetweenWords
  {\hskip3\dimexpr\MorseWidth\relax}

\def\MorseSpace
  {\hskip7\dimexpr\MorseWidth\relax}

\def\MorseUnknown#1
  {[\detokenize{#1}]}
```

The dimensions are stored in macros as well. Of course we could provide a proper setup command, but it hardly makes sense.

```
\def\MorseWidth {0.4em}
\def\MorseHeight{0.2em}
```

Finally we have arrived at the macros that interface to the LUA functions.

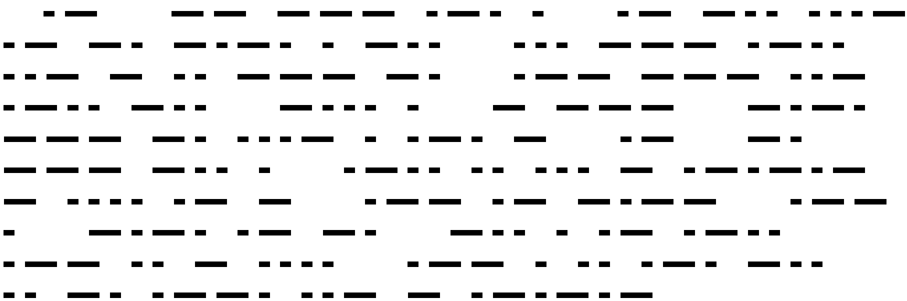
```
\def\MorseString#1{\ctxlua{moduledata.morse.tomorse(\!!bs#1\!!es)}}
\def\MorseFile  #1{\ctxlua{moduledata.morse.filetomorse("#1")}}
```

```
\def\MorseTable   {\ctxlua{moduledata.morse.showtable()}}
```

A string is converted to morse with the first command.

```
\Morse{A more advanced solution would be to convert a node list. That
way we can deal with weird input.}
```

This shows up as:



Reduction and uppercasing is demonstrated in the next example:

```
\MorseString{ÀÁÂÃÄÅàáâãäå}
```

This gives:

Using helpers

The next example shows a bit of LPEG. On top of the standard functionality a few additional functions are provided. Let's start with a pure TEX example:

```
\defineframed
  [colored]
  [foregroundcolor=red,
   foregroundstyle=\underbar,
   offset=.1ex,
   location=low]
\processisolatedwords {\input ward \relax} \colored
```

that results in: The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day — and we humans are the cigarettes.

Because this processor macro operates at the TEX end it has some limitations. The content is collected in a very narrow box and from that a regular paragraph is constructed. It is for this reason that no color is applied: the snippets that end up in the box are already typeset.

An alternative is to delegate the task to LUA:

```
\startluacode
local function process(data)

  local words = lpeg.split(lpeg.patterns.spacer,data or "")

  for i=1,#words do
    if i == 1 then
        context.dontleavehmode()
    else
        context.space()
    end
    context.colored(words[i])
  end

end

process(io.loaddata(resolvers.findfile("ward.tex")))
\stopluacode
```

This results in: The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day — and we humans are the cigarettes.

The function splits the loaded data into a table with individual words. We use a splitter that splits on spacing tokens. The special case for `i = 1` makes sure that we end up in horizontal mode (read: properly start a paragraph). This time we do get color because the typesetting is done directly. Here is an alternative implementation:

```
local done = false

local function reset()
    done = false
    return true
end
local function apply(s)
  if done then
    context.space()
  else
    done = true
    context.dontleavehmode()
```

```
  end
  context.colored(s)
end

local splitter = lpeg.P(reset)
               * lpeg.splitter(lpeg.patterns.spacer,apply)

local function process(data)
  lpeg.match(splitter,data)
end
```

This version is more efficient as it does not create an intermediate table. The next one is comparable:

```
local function apply(s)
  context.colored("%s ",s)
end

local splitter lpeg.splitter(lpeg.patterns.spacer,apply)

local function process(data)
  context.dontleavevmode()
  lpeg.match(splitter,data)
  context.removeunwantedspaces()
end
```

Formatters

Sometimes you can save a bit of work by using formatters. By default, the `context` command, when called directly, applies a given formatter. But when called as table this feature is lost because then we want to process non-strings as well. The next example shows a way out:

```
context("the current emwidth is %p", \number\emwidth)
context.par()
context.formatted("the current emwidth is %p", \number\emwidth)
context.par()
context.bold(string.formatters["the current emwidth is %p"](
  \number\emwidth))
context.par()
context.formatted.bold("the current emwidth is %p",
  \number\emwidth)
```

The last one is the most interesting one here: in the subnamespace `formatted` (watch the `d`) a format specification with extra arguments is expected. This is the result: the current emwidth is 10pt

the current emwidth is 10pt
the current emwidth is 10pt
the current emwidth is 10pt.

Summary

context("...")
The string is flushed directly.

```
...
```

context("format",...)
The first string is a format specification according that is passed to the LUA function `format` in the `string` namespace. Following arguments are passed too.

```
format("format",...)
```

context(123,...)
The numbers (and following numbers or strings) are flushed without any formatting.

```
123... (concatenated)
```

context(true)
An explicit `endlinechar` is inserted.

```
^^M
```

context(false,...)
Strings and numbers are flushed surrounded by curly braces, an indexed table is flushed as option list, and a hashed table is flushed as parameter set.

```
multiple {...} or [...] etc
```

context(node)
The node(list) is injected at the spot. Keep in mind that you need to do the proper memory management yourself.

context.command(value,...)
The value (string or number) is flushed as a curly braced (regular) argument.

`\command {value}...`

context.command(value ,...)
The table is flushed as value set. This can be an identifier, a list of options, or a directive.
`\command [value]...`

context.command(key = value ,...)
The table is flushed as key/value set.
`\command [key={value}]...`

context.command(true)
An explicit `endlinechar` is inserted.
`\command ^^M`

context.command(node)
The node(list) is injected at the spot. Keep in mind that you need to do the proper memory management yourself.
`\command {node(list)}`

context.command(false,value)
The value is flushed without encapsulating tokens.
`\command value`

context.command(value , key = value , value, false, value)
The arguments are flushed accordingly their nature and the order can be any.
`\command [value][key={value}]{value}value`

context.direct(...)
The arguments are interpreted the same as if `direct` was a command, but no `\direct` is injected in front.

context.delayed(...)
The arguments are interpreted the same as in a `context` call, but instead of a direct flush, the arguments will be flushed in a next cycle.

context.delayed.command(...)
The arguments are interpreted the same as in a `command` call, but instead of a direct flush, the command and arguments will be flushed in a next cycle.

context.nested.command
This command returns the command, including given arguments as a string. No flushing takes place.

context.nested
This command returns the arguments as a string and treats them the same as a regular `context` call.

context.formatted.command
This command returns the command that will pass it's arguments to the string formatter.

context.formatted
This command passes it's arguments to the string formatter.

context.metafun.start(...)
This starts a MetaFun (or MetaPost) graphic.

context.metafun()
This finishes and flushes a MetaFun (or MetaPost) graphic.

context.metafun.stop(...)
The argument is appended to the current graphic data.

context.metafun.stop("format",...)
The argument is appended to the current graphic data but the string formatter is used on following arguments.

ConTEXt–Lua dokumenty

V rámci formátu ConTEXt lze připravovat dokumenty pomocí kombinace jazyků TEX, xml, MetaPost a Lua. Článek v krátkosti shrnuje základy jazyka Lua a následně se věnuje způsobům, jakými je možné jazyk využít při přípravě dokumentů ve formátu ConTEXt MkIV.

Klíčová slova: Lua, LuaTEX, ConTEXt, MkIV

Hans Hagen, pragma@wxs.nl