ConTEXt Lua Documents

Hans Hagen

Contents

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 chapters 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. So, keep reading,

Hans Hagen
Hasselt NL
2009–2010

1 A bit of Lua

1.1 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. Go out and 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.

1.2 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 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 cross platform identical, so we don't need to worry about compatibility.

Strings in Lua can be given between quotes or can be so called long strings.

```
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 = s
local u = t
```

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 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
n == nil    true
```

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.

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), `bit` (in recent versions). 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.

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
```

1.3 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") -- sp or 'scaled point' is the smallest unit
```

The unit of dimension is ‘scaled point’ and this is a pretty small unit: 10 points equal 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])
```

1.4 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
```

Here the function call to `pairs(list)` returns `next, list` so there is an 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.

1.5 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.

1.6 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.[1] 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=ipad
```

todo: add nice synonym for this module and also add helpinfo at the to so that we can do `context --styles`

1.7 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 and span multiple lines]]
```

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

[1] 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.

2 Getting started

2.1 Some basics

To start with, I assume that you have either the so called ConTEXt 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` 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 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, that is, 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.

2.2 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 backets (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}
\startchapter[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 whilc the second category normally concerns some text to be typeset.

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.

2.3 Spaces and Lines

In a regular TEX file, spaces and newline characters are collapsed into one space. At the Lua end the same happens. Compare the following examples. First we omit spaces:

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

leftmiddleright

Next we add spaces:

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

left middle right

We can also add more spaces:

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

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")
```

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")
```

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")
```

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()
```

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.

2.4 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 have 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()
```

2.5 Catcodes

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

```
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 of course 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")
```

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()
```

$\pi = 3.14159$

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 call for 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.

3 More on functions

3.1 Why we need them

In a previous chapter 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.

3.2 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.

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 explicltly, 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.

3.3 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 *any* 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`.

4 A few Details

4.1 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.

4.2 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.

4.3 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 `Don't get framed!` as all tokens are considered to be letters.

4.4 Node lists

If you're not deep into TeX you will never feel the need to manipulate nodelists 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.

5 Some more examples

5.1 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 5.1**. The actual cover page was derived from this.

5.2 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()
```

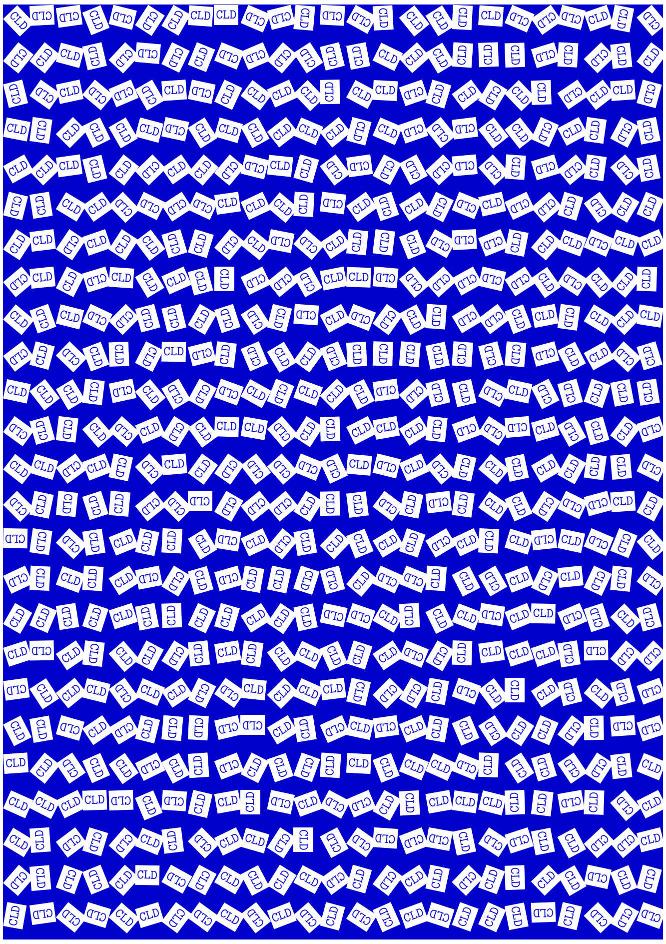


Figure 5.1 The simplified cover page.

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",
```

32	17	77	23	60	12	53	21	49	91	73	80	9	16	47	65	4	58	86	79
72	25	80	85	61	94	78	41	56	1	51	53	12	5	94	51	49	60	42	96
41	26	46	10	45	23	8	40	5	98	1	60	77	99	73	12	17	93	86	73
61	65	46	3	55	13	13	92	57	79	39	4	71	39	1	41	37	74	6	89
22	61	19	4	67	8	92	75	5	52	48	50	11	72	97	32	14	14	42	41
25	74	44	66	98	82	9	81	91	70	67	58	47	3	5	49	47	65	16	70
73	53	40	72	25	79	58	7	16	43	94	83	18	70	12	46	57	71	47	27
47	25	73	33	48	41	22	27	46	4	73	30	46	44	39	30	56	58	35	31
70	6	86	96	32	8	48	47	64	88	95	62	67	69	55	39	71	63	37	60
25	70	49	27	28	42	26	7	82	99	87	53	62	69	37	47	90	39	16	13

Table 5.1 A table generated by Lua.

```
  background = "color",
  backgroundcolor = "darkblue",
  foregroundcolor = "white",
}
local random = math.random
context.startlinecorrection { "blank" }
  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()
context.stoplinecorrection()
```

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 5.1**. The 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.

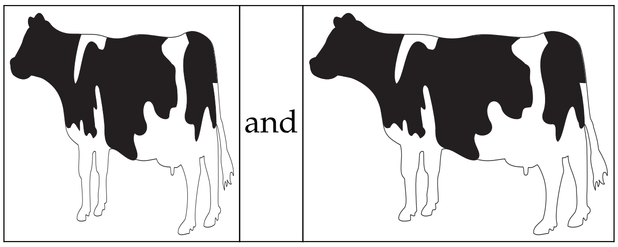


Figure 5.2 Two cows!

5.3 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 !")
```

This is **important**, and **this** too !

5.4 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 calculus 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 calculus. 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
```

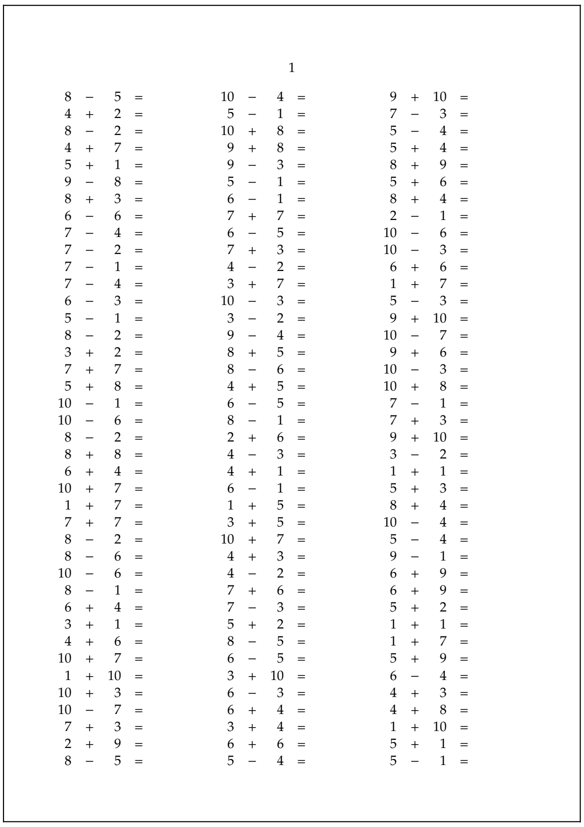
This is a typical example of where it's more convenient to write the code in Lua that in TEX's macro language. As a consequence setting up the page also happens in Lua:

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

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

This leave 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()
```

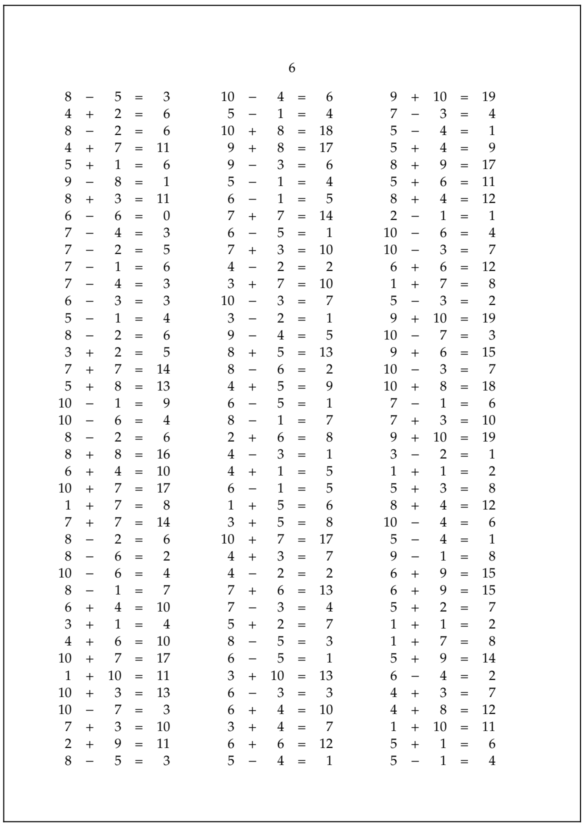
```
  ForLorien(n,10,10,true)
  ForLorien(n,20,20,true)
  ForLorien(n,30,30,true)
  ForLorien(n,40,40,true)
  ForLorien(n,50,50,true)

context.stoptext()
```



1

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

exercises



6

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

answers

Figure 5.3 Lorien's challenge.

A few pages of the result are shown in **figure 5.3**. In the ConTEXt distribution more advanced version can be found in `s-edu-01.cld` as I was also asked to generate multiplication and table exercises. I also 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.calculus.generate {
  name     = "Bram Otten",
  fontsize = "12pt",
  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 },
```

```
  },
}
```

6 Graphics

6.1 The regular interface

If you are familiar with ConTEXt, which by now probably is the case, you will have noticed that it integrates the MetaPost graphic subsystem. Drawing a graphic is not that complex:

```
context.startMPcode()
context [[
  draw
    fullcircle scaled 1cm
    withpen pencircle scaled 1mm
    withcolor .5white
    dashed dashpattern (on 2mm off 2mm) ;
 ]]
context.stopMPcode()
```

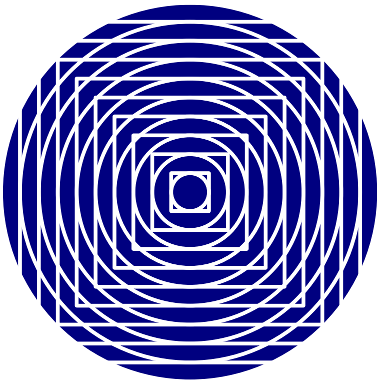
We get a gray dashed circle rendered with an one millimeter thick line:

So, we just use the regular commands and pass the drawing code as strings. Although MetaPost is a rather normal language and therefore offers loops and conditions and the lot, you might want to use Lua for anything else than the drawing commands. Of course this is much less efficient, but it could be that you don't care about speed. The next example demonstrates the interface for building graphics piecewise.

```
context.resetMPdrawing()

context.startMPdrawing()
context([[fill fullcircle scaled 5cm withcolor (0,0,.5) ;]])
context.stopMPdrawing()

context.MPdrawing("pickup pencircle scaled .5mm ;")
context.MPdrawing("drawoptions(withcolor white) ;")

for i=0,50,5 do
  context.startMPdrawing()
  context("draw fullcircle scaled %smm ;",i)
  context.stopMPdrawing()
end

for i=0,50,5 do
  context.MPdrawing("draw fullsquare scaled " .. i .. "mm ;")
end
```

```
context.MPdrawingdonetrue()

context.getMPdrawing()
```

This gives:



I the first loop we can use the format options associated with the simple `context` call. This will not work in the second case. Even worse, passing more than one argument will definitely give a faulty graphic definition. This is why we have a special interface for MetaFun. The code above can also be written as:

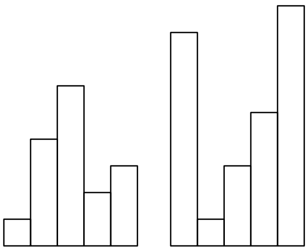
```
local metafun = context.metafun

metafun.start()

metafun("fill fullcircle scaled 5cm withcolor %s ;",
    metafun.color("darkblue"))

metafun("pickup pencircle scaled .5mm ;")
metafun("drawoptions(withcolor white) ;")

for i=0,50,5 do
  metafun("draw fullcircle scaled %smm ;",i)
end

for i=0,50,5 do
  metafun("draw fullsquare scaled %smm ;",i)
end

metafun.stop()
```

Watch the call to `color`, this will pass definitions at the TEX end to MetaPost. Of course you really need to ask yourself “Do I want to use MetaPost this way?”. Using Lua loops instead of MetaPost ones makes much more sense in the following case:

```
local metafun = context.metafun

function metafun.barchart(t)
```

```
  metafun.start()
  local t = t.data
  for i=1,#t do
    metafun("draw unitsquare xyscaled(%s,%s) shifted (%s,0);",
      10, t[i]*10, i*10)
  end
  metafun.stop()
end

local one = { 1, 4, 6, 2, 3, }
local two = { 8, 1, 3, 5, 9, }

context.startcombination()
  context.combination(metafun.delayed.barchart { data = one }, "one")
  context.combination(metafun.delayed.barchart { data = two }, "two")
context.stopcombination()
```

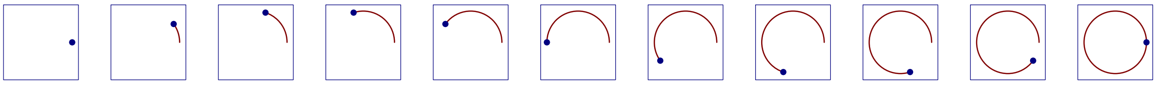
We get two barcharts alongside:



one two

```
local template = [[
  path p, q ; color c[] ;
  c1 := \MPcolor{darkblue} ;
  c2 := \MPcolor{darkred} ;
  p := fullcircle scaled 50 ;
  l := length p ;
  n := %s ;
  q := subpath (0,%s/n*l) of p ;
  draw q withcolor c2 withpen pencircle scaled 1 ;
  fill fullcircle scaled 5 shifted point length q of q withcolor c1 ;
  setbounds currentpicture to unitsquare shifted (-0.5,-0.5) scaled 60 ;
  draw boundingbox currentpicture withcolor c1 ;
  currentpicture := currentpicture xsized(1cm) ;
]]

local function steps(n)
  for i=0,n do
    context.metafun.start()
      context.metafun(template,n,i)
    context.metafun.stop()
```

```
    if i < n then
        context.quad()
    end
  end
end

context.hbox(function() steps(10) end)
```



To some extent we fool ourselves with this kind of Luafication of MetaPost code. Of course we can make a nice MetaPost library and put the code in a macro instead. In that sense, doing this in ConTEXt directly often gives better and more efficient code.

Of course you can use all relevant commands in the Lua interface, like:

```
context.startMPpage()
  context("draw origin")
  for i=0,100,10 do
    context("..{down}(%d,0)",i)
  end
  context(" withcolor \\MPcolor{darkred} ;")
context.stopMPpage()
```

to get a graphic that has its own page. Don't use the `metafun` namespace here, as it will not work here. This drawing looks like:

6.2 The Lua interface

todo

7 Verbatim

7.1 Introduction

If you are familiar with traditional TEX, you know that some characters have special meanings. For instance a `$` starts and ends inline math mode:

```
$e=mc^2$
```

If we want to typeset math from the Lua end, we can say:

```
context.mathematics("e=mc^2")
```

This is in fact:

```
\mathematics{e=mc^2}
```

However, if we want to typeset a dollar and use the `ctxcatcodes` regime, we need to explicitly access that character using `\char` or use a command that expands into the character with catcode other.

One step further is that we typeset all characters as they are and this is called verbatim. In that mode all characters are tokens without any special meaning.

7.2 Special treatment

The formula in the introduction can be typeset verbatim as follows:

```
context.verbatim("$e=mc^2$")
```

This gives:

$e=mc^2$

You can also do things like this:

```
context.verbatim.bold("$e=mc^2$")
```

Which gives:

$e=mc^2$

So, within the `verbatim` namespace, each command gets its arguments verbatim.

```
context.verbatim.inframed({ offset = "0pt" }, "$e=mc^2$")
```

Here we get: $e=mc^2$. So, settings and alike are processed as if the user had used a regular `context.inframed` but the content comes out verbose.

If you wonder why verbatim is needed as we also have the `type` function (macro) the answer is that it is faster, easier to key in, and sometimes needed.

7.3 Multiple lines

Currently we have to deal with linebreaks in a special way. This is due to the way TEX deals with linebreaks. In fact, when we print something to TEX, the text after a `\n` is simply ignored.

For this reason we have a few helpers. If you want to put something in a buffer, you cannot use the regular buffer functions unless you make sure that they are not overwritten while you're still at the Lua end.

```
context.tobuffer("temp",str)
context.getbuffer("temp")
```

Another helper is the following. It splits the string into lines and feeds them piecewise using the `context` function and in the process adds a space at the end of the line (as this is what TEX normally does.

```
context.tolines(str)
```

7.4 Pretty printing

In ConTEXt MkII there have always been pretty printing options. We needed it for manuals and it was also handy to print sources in the same colors as the editor uses. Most of those pretty printers work in a line-by-line basis, but some are more complex, especially when comments or strings can span multiple lines.

When the first versions of LuaTEX showed up, rewriting the MkII code to use Lua was a nice exercise and the code was not that bad, but when lpeg showed up, I put it on the agenda to reimplement them again.

We only ship a few pretty printers. Users normally have their own preferences and it's not easy to make general purpose pretty printers. This is why the new framework is a bit more flexible and permits users to kick in their own code.

Pretty printing involves more than coloring some characters or words:

- spaces should honoured and can be visualized
- newlines and empty lins need to be honoured as well
- optionally lines have to be numbered but
- wrapped around lines should not be numbered

It's not much fun to deal with these matters each time that you write a pretty printer. This is why we can start with an existing one like the default pretty printer. We show several variants of doing the same. We start with a simple clone of the default parser.

```
local P, V = lpeg.P, lpeg.V
```

```
local grammar = visualizers.newgrammar("default", {
  pattern    = V("default:pattern"),
  visualizer = V("pattern")^1
} )

local parser = P(grammar)

visualizers.register("test-0", { parser = parser })
```

We distinguish between grammars (tables with rules), parsers (a grammar turned into an lpeg expression), and handlers (collections of functions that can be applied. All three are registered under a name and the verbatim commands can refer to that name.

```
\starttyping[option=test-0,color=]
Test 123,
test 456 and
test 789!
\stoptyping
```

Nothing special happens here. We just get straightforward verbatim.

```
Test 123,
test 456 and
test 789!
```

Next we are going to color digits. We collect as many as possible in a row, so that we minimize the calls to the colorizer.

```
local patterns, P, V = lpeg.patterns, lpeg.P, lpeg.V

local function colorize(s)
  context.color{"darkred"}
  visualizers.writeargument(s)
end

local grammar = visualizers.newgrammar("default", {
  digit      = patterns.digit^1 / colorize,
  pattern    = V("digit") + V("default:pattern"),
  visualizer = V("pattern")^1
} )

local parser = P(grammar)

visualizers.register("test-1", { parser = parser })
```

Watch how we define a new rule for the digits and overload the pattern rule. We can refer to the default rule by using a prefix. This is needed when we define a rule with the same name.

```
\starttyping[option=test-1,color=]
```

```
Test 123,
test 456 and
test 789!
\stoptyping
```

This time the digits get colored.

```
Test 123,
test 456 and
test 789!
```

In a similar fashion we can colorize letters. As with the previous example, we use ConTEXt commands at the Lua end.

```
\starttyping[option=test-2,color=]
Test 123,
test 456 and
test 789!
\stoptyping
```

Again we get some coloring.

```
Test 123,
test 456 and
test 789!
```

It will be clear that the amount of rules and functions is larger when we use a more complex parser. It is for this reason that we can group functions in handlers. We can also make a pretty printer configurable by defining handlers at the TEX end.

```
\definestartstop
  [MyDigit]
  [style=bold,color=darkred]

\definestartstop
  [MyLowercase]
  [style=bold,color=darkgreen]

\definestartstop
  [MyUppercase]
  [style=bold,color=darkblue]
```

The Lua code now looks different. Watch out. We need an indirect call to for instance `MyDigit` because a second argument can be passed: the settings for this environment and you don't want that get passed to `MyDigit` and friends.

```
\starttyping[option=test-3,color=]
Test 123,
test 456 and
```

```
test 789!
\stoptyping
```

We get digits, upper- and lowercase characters colored:

```
Test 123,
test 456 and
test 789!
```

You can also use parsers that don't use lpeg:

```
local function parser(s)
  visualizers.write("["..s.."]")
end

visualizers.register("test-4", { parser = parser })

\starttyping[option=test-4,space=on,color=darkred]
Test 123,
test 456 and
test 789!
\stoptyping
```

The function `visualizer.write` takes care of spaces and newlines.

```
[Test␣123,
test␣456␣and
test␣789!]
```

We have a few more helpers:

`visualizers.write`	interprets the argument and applies methods
`visualizers.writenewline`	goes to the next line (similar to `\par`
`visualizers.writeemptyline`	inserts an empty line (similer to `\blank`
`visualizers.writespace`	inserts a (visible) space
`visualizers.writedefault`	writes the argument verbatim without interpretation

These mechanism have quite some overhead in terms of function calls. In the worst case each token needs a (nested) call. However, doing all this at the TEX end also comes at a price. So, in practice this approach is more flexible but without too large a penalty.

In all these examples we typeset the text verbose: what is keyed in normally comes out (either or not with colors), so spaces stay spaces and linebreaks are kept.

```
local function parser(s)
  local s = string.gsub(s,"show","demonstrate")
  local s = string.gsub(s,"'re"," are")
  context(s)
end
```

```
visualizers.register("test-5", { parser = parser })
```

We can apply this visualizer as follows:

```
\starttyping[option=test-5,color=darkred,style=]
This is just some text to show what we can do with this mechanism. In
spite of what you might think we're not bound to verbose text.
\stoptyping
```

This time the text gets properly aligned:

This is just some text to demonstrate what we can do with this mechanism. In spite of what you might think we are not bound to verbose text.

It often makes sense to use a buffer:

```
\startbuffer[demo]
This is just some text to show what we can do with this mechanism. In
spite of what you might think we're not bound to verbose text.
\stopbuffer
```

Instead of processing the buffer in verbatim mode you can then process it directly:

```
\setuptyping[file][option=test-5,color=darkred,style=]
\processbuffer[demo]
```

Which gives:

This is just some text to demonstrate what we can do with this mechanism. In spite of what you might think we are not bound to verbose text.

In this case, the space is a normal space and the fixed verbatim space, which looks nicer.

8 Lua Functions

8.1 Introduction

When you run ConTEXt you have some libraries preloaded. If you look into the Lua files you will more than is discussed here, but keep in mind that what is not documented, might be gone or done different one day. Some extensions live in the same namespace as those provided by stock Lua and LuaTEX, others have their own. There are many more functions and the more obscure (or never being used) ones will go away.

The Lua code in ConTEXt is organized in quite some modules. Those with names like `l-*.lua` are rather generic and are automatically available when you use `mtxrun` to run a Lua file. These are discusses in this chapter. A few more modules have generic properties, like some in the categories `util-*.lua`, `trac-*.lua`, `luat-*.lua`, `data-*.lua` and `lxml-*.lua`. They contain more specialized functions and are discussed elsewhere.

8.2 Tables

[lua] concat

These functions come with Lua itself and are discussed in detail in the Lua reference manual so we stick to some examples. The `concat` function stitches table entries in an indexed table into one string, with an optional separator in between. If can also handle a slice of the table

```
local str = table.concat(t)
local str = table.concat(t,separator)
local str = table.concat(t,separator,first)
local str = table.concat(t,separator,first,last)
```

Only strings and numbers can be concatenated.

table.concat({"a","b","c","d","e"})

```
abcde
```

table.concat({"a","b","c","d","e"},"+")

```
a+b+c+d+e
```

table.concat({"a","b","c","d","e"},"+",2,3)

```
b+c
```

[lua] insert remove

You can use `insert` and `remove` for adding or replacing entries in an indexed table.

```
table.insert(t,value,position)
value = table.remove(t,position)
```

The position is optional and defaults to the last entry in the table. For instance a stack is built this way:

```
table.insert(stack,"top")
local top = table.remove(stack)
```

Beware, the `insert` function returns nothing. You can provide an additional position:

```
table.insert(list,"injected in slot 2",2)
local thiswastwo = table.remove(list,2)
```

[lua] unpack

You can access entries in an indexed table as follows:

```
local a, b, c = t[1], t[2], t[3]
```

but this does the same:

```
local a, b, c = table.unpack(t)
```

This is less efficient but there are situations where `unpack` comes in handy.

[lua] sort

Sorting is done with `sort`, a function that does not return a value but operates on the given table.

```
table.sort(t)
table.sort(t,comparefunction)
```

The compare function has to return a consistent equivalent of `true` or `false`. For sorting more complex data structures there is a specialized sort module available.

t={"a","b","c"} table.sort(t)

```
t={  "a",  "b",  "c", }
```

t={"a","b","c"} table.sort(t,function(x,y) return x > y end)

```
t={  "c",  "b",  "a", }
```

t={"a","b","c"} table.sort(t,function(x,y) return x < y end)

```
t={  "a",  "b",  "c", }
```

keys sortedkeys sortedhashkeys sortedhash

The `keys` function returns an indexed list of keys. The order is undefined as it depends on how the table was constructed. A sorted list is provided by `sortedkeys`. This function is rather liberal with respect to the keys. If the keys are strings you can use the faster alternative `sortedhashkeys`.

```
local s = table.keys (t)
local s = table.sortedkeys (t)
local s = table.sortedhashkeys (t)
```

Because a sorted list is often processed there is also an iterator:

```
for key, value in table.sortedhash(t) do
    print(key,value)
end
```

There is also a synonym `sortedpairs` which sometimes looks more natural when used alongside the `pairs` and `ipairs` iterators.

table.keys({ [1] = 2, c = 3, [true] = 1 })

```
t={  1,  true,  "c", }
```

table.sortedkeys({ [1] = 2, c = 3, [true] = 1 })

```
t={  1,  "c",  true, }
```

table.sortedhashkeys({ a = 2, c = 3, b = 1 })

```
t={  "a",  "b",  "c", }
```

serialize print tohandle tofile

The `serialize` function converts a table into a verbose representation. The `print` function does the same but prints the result to the console which is handy for tracing. The `tofile` function writes the table to a file, using reasonable chunks so that less memory is used. The fourth variant `tohandle` takes a handle so that you can do whatever you like with the result.

```
table.serialize (root, name, reduce, noquotes, hexify)
table.print (root, name, reduce, noquotes, hexify)
table.tofile (filename, root, name, reduce, noquotes, hexify)
table.tohandle (handle, root, name, reduce, noquotes, hexify)
```

The serialization can be controlled in several ways. Often only the first two options makes sense:

table.serialize({ a = 2 })

```
t={  ["a"]=2, }
```

table.serialize({ a = 2 }, "name")

```
name={  ["a"]=2, }
```

table.serialize({ a = 2 }, true)

```
return {  ["a"]=2, }
```

table.serialize({ a = 2 }, false)

```
{  ["a"]=2, }
```

table.serialize({ a = 2 }, "return")

```
return {  ["a"]=2, }
```

table.serialize({ a = 2 }, 12)

```
[12]={  ["a"]=2, }
```

table.serialize({ a = 2, [3] = "b", [true] = "6" }, nil, true)

```
t={  [3]="b",  ["a"]=2,  [true]=6, }
```

table.serialize({ a = 2, [3] = "b", [true] = "6" }, nil, true, true)

```
t={  [3]="b",  a=2,  [true]=6, }
```

table.serialize({ a = 2, [3] = "b", [true] = "6" }, nil, true, true, true)

```
t={  [0x0003]="b",  a=0x0002,  [true]=6, }
```

In ConTEXt there is also a `tocontext` function that typesets the table verbose. This is handy for manuals and tracing.

identical are_equal

These two function compare two tables that have a similar structure. The `identical` variant operates on a hash while `are_equal` assumes an indexed table.

```
local b = table.identical (one, two)
local b = table.are_equal (one, two)
```

table.identical({ a = { x = 2 } }, { a = { x = 3 } })

```
false
```

table.identical({ a = { x = 2 } }, { a = { x = 2 } })

```
true
```

table.are_equal({ a = { x = 2 } }, { a = { x = 3 } })

```
true
```

table.are_equal({ a = { x = 2 } }, { a = { x = 2 } })

```
true
```

table.identical({ "one", "two" }, { "one", "two" })

```
true
```

table.identical({ "one", "two" }, { "two", "one" })

```
false
```

table.are_equal({ "one", "two" }, { "one", "two" })

```
true
```

table.are_equal({ "one", "two" }, { "two", "one" })

```
false
```

tohash fromhash swapped swaphash reverse

We use `tohash` quite a lot in ConTEXt. It converts a list into a hash so that we can easily check if (a string) is in a given set. The `fromhash` function does the opposite: it creates a list of keys from a hashed table where each value that is not `false` or `nil` is present.

```
local hashed  = table.tohash  (indexed)
local indexed = table.fromhash(hashed)
```

The function `swapped` turns keys into values and reverse while the `reverse` function reverses the values in an indexed table.

```
local swapped  = table.swapped  (indexed)
local reversed = table.reversed (indexed)
```

table.tohash({ "a", "b", "c" })

```
t={  ["a"]=true,  ["b"]=true,  ["c"]=true, }
```

table.fromhash({ a = true, b = false, c = true })

```
t={  "a",  "c", }
```

table.swapped({ "a", "b", "c" })

```
t={  ["a"]=1,  ["b"]=2,  ["c"]=3, }
```

table.reversed({ "a", "b", "c" })

```
t={  "c",  "b",  "a", }
```

append prepend

These two functions operate on a pair of indexed tables. The first table gets appended or prepended by the second. The first table is returned as well.

```
table.append (one, two)
table.prepend(one, two)
```

The functions are similar to loops using `insert`.

table.append({ "a", "b", "c" }, { "d", "e" })

```
t={  "a",  "b",  "c",  "d",  "e", }
```

table.prepend({ "a", "b", "c" }, { "d", "e" })

```
t={  "d",  "e",  "a",  "b",  "c", }
```

merge merged imerge imerged

You can merge multiple hashes with `merge` and indexed tables with `imerge`. The first table is the target and is returned.

```
table.merge   (one, two, ...)
table.imerge  (one, two, ...)
```

The variants ending with a `d` merge the given list of tables and return the result leaving the first argument untouched.

```
local merged = table.merged  (one, two, ...)
local merged = table.imerged (one, two, ...)
```

table.merge({ a = 1, b = 2, c = 3 }, { d = 1 }, { a = 0 })

```
t={  ["a"]=0,  ["b"]=2,  ["c"]=3,  ["d"]=1, }
```

table.imerge({ "a", "b", "c" }, { "d", "e" }, { "f", "g" })

```
t={  "a",  "b",  "c",  "d",  "e",  "f",  "g", }
```

copy fastcopy

When copying a table we need to make a real and deep copy. The `copy` function is an adapted version from the Lua wiki. The `fastopy` is faster because it does not check for circular references and does not share tables when possible. In practice using the fast variant is okay.

```
local copy = table.copy    (t)
local copy = table.fastcopy(t)
```

flattened

A nested table can be unnested using `flattened`. Normally you will only use this function if the content is somewhat predictable. Often using one of the merge functions does a similar job.

```
local flattened = table.flatten(t)
```

table.flattened({ a =1, b = 2, { c = 3 }, d = 4})

```
t={  ["a"]=1,  ["b"]=2,  ["c"]=3,  ["d"]=4, }
```

table.flattened({ 1, 2, { 3, { 4 } }, 5})

```
t={  1,  2,  3,  4,  5, }
```

table.flattened({ 1, 2, { 3, { 4 } }, 5}, 1)

```
t={  1,  2,  3,  { 4 },  5, }
```

table.flattened({ a = 1, b = 2, { c = 3 }, d = 4})

```
t={  ["a"]=1,  ["b"]=2,  ["c"]=3,  ["d"]=4, }
```

table.flattened({ 1, 2, { 3, { c = 4 } }, 5})

```
t={  1,  2,  3,  5,  ["c"]=4, }
```

table.flattened({ 1, 2, { 3, { c = 4 } }, 5}, 1)

```
t={  1,  2,  3,  {  ["c"]=4,  },  5, }
```

contains

This function works with indexed tables. Watch out, when you look for a match, the number `1` is not the same as string `"1"`. The function returns the index or `false`.

```
if table.contains(t, 5 ) then ... else ... end
if table.contains(t,"5") then ... else ... end
```

table.contains({ "a", 2, true, "1"}, 1)

```
false
```

table.contains({ "a", 2, true, "1"}, "1")

```
4
```

count

The name speaks for itself: this function counts the number of entries in the given table. For an indexed table `#t` is faster.

```
local n = table.count(t)
```

table.count({ 1, 2, [4] = 4, a = "a" })

```
4
```

sequenced

Normally, when you trace a table, printing the serialized version is quite convenient. However, when it concerns a simple table, a more compact variant is:

```
print(table.sequenced(t, separator))
```

table.sequenced({ 1, 2, 3, 4})

```
1=1 | 2=2 | 3=3 | 4=4
```

table.sequenced({ 1, 2, [4] = 4, a = "a" }, ", ")

```
1=1, 2=2, 4=4, a=a
```

8.3 Math

In addition to the built-in math function we provide: `round`, `div`, `mod`, `sind`, `cosd` and `tand`.

8.4 Booleans

tonumber

This function returns the number one or zero. You will seldom need this function.

```
local state = boolean.tonumber(str)
```

boolean.tonumber(true)

```
1
```

toboolean

When dealing with configuration files or tables a bit flexibility in setting a state makes sense, if only because in some cases it's better to say `yes` than `true`.

```
local b = toboolean(str)
local b = toboolean(str,tolerant)
```

When the second argument is true, the strings `true`, `yes`, `on`, `1`, `t` and the number `1` all turn into `true`. Otherwise only `true` is honoured. This function is also defined in the global namespace.

string.toboolean("true")

```
true
```

string.toboolean("yes")

```
yes
```

string.toboolean("yes",true)

```
true
```

is_boolean

This function is somewhat similar to the previous one. It interprets the strings `true`, `yes`, `on` and `t` as `true` and `false`, `no`, `off` and `f` as `false`. Otherwise `nil` is returned, unless a default value is given, in which case that is returned.

```
if is_boolean(str)         then ... end
if is_boolean(str,default) then ... end
```

string.is_boolean("true")

```
true
```

string.is_boolean("off")

```
false
```

string.is_boolean("crap",true)

```
true
```

8.5 Strings

Lua strings are simply sequences of bytes. Of course in some places special treatment takes place. For instance `\n` expands to one or more characters representing a newline, depending on the operating system, but normally, as long as you manipulate strings in the perspective of LuaTEX, you don't need to worry about such issues too much. As LuaTEX is a utf-8 engine, strings normally are in that encoding but again, it does not matter much. First of all we have the `unicode` library linked into LuaTEX, but most of all, Lua is quite agnostic about the content of strings: it does not care about three characters reflecting one Unicode character or not. This means that when you use for instance the functions discussed here, or use libraries like `lpeg` behave as you expect.

[lua] sub

You cannot directly access a character in a string but you can take any slice you want using `sub`. You need to provide a start position and negative values will count backwards from the end.

```
local slice = string.sub(str,first,last)
```

string.sub("abcdef",2)

```
bcdef
```

string.sub("abcdef",2,3)

```
bc
```

string.sub("abcdef",-3,-2)

```
de
```

[lua] gsub

There are two ways of analyzing the content of a string. The more modern and flexible approach is to use `lpeg`. The other one uses some functions in the `string` namespace that accept so called patterns for matching. While `lpeg` is more powerfull than regular expressions, the pattern matching is less powerfull but sometimes faster and also easier to specify. In many cases it can do the job quite well.

```
local new, count = string.gsub(old,pattern,replacement)
```

The replacement can be a function. Often you don't want the number of matches, and the way to avoid this is either to store the result in a variable:

```
local new = string.gsub(old,"lua","LUA")
print(new)
```

or to use parentheses to signal the interpreter that only one value is return.

```
print((string.gsub(old,"lua","LUA"))
```

Patterns can be more complex so you'd better read the Lua manual if you want to know more about them.

string.gsub("abcdef","b","B")

```
aBcdef
```

string.gsub("abcdef","[bc]",string.upper)

```
aBCdef
```

[lua] find

The `find` function returns the first and last position of the match:

```
local first, last = find(str,pattern)
```

If you're only interested if there is a match at all, it's enough to know that there is a first position. No match returns `nil`. So,

```
if find("luatex","tex") then ... end
```

works out okay. You can pass an extra argument to `find` that indicates the start position. So you can use this function to loop over all matches: just start again at the end of the last match.

[lua] match gmatch

With `match` you can split of bits and pieces of a string. The parenthesis indicate the captures.

```
local a, b, c, ... = string.match(str,pattern)
```

The `gmatch` function is used to loop over a string, for instance the following code prints the elements in a comma separated list, ignoring spaces after commas.

```
for s in string.gmatch(str,"([^,%s])+") do
  print(s)
end
```

string.match("before:after","^(.-):")

```
before
```

[lua] lower upper

These two function spreak for themselves.

string.lower("LOW")

```
low
```

string.upper("upper")

```
UPPER
```

[lua] format

The `format` function takes a template as first argument and one or more additional arguments depending on the format. The template is similar to the one used in c but it has some extensions.

```
local s = format(format, str, ...)
```

string.format("U+%05X",2010)

```
U+007DA
```

[luatex] utfvalues utfcharacters

There are a couple of extra functions implemented in LuaTEX that deal with utf. The following function loops over the utf characters in a string and returns the Unicode number in `u`:

```
for u in utf.utfvalues(str) do
    ... -- u is a number
end
```

The next one returns a string `c` that has one or more characters as utf characters can have upto 4 bytes.

```
for c in utf.utfcharacters(str) do
    ... -- c is a string
end
```

strip

This function removes any leading and trailing whitespace characters.

```
local s = string.strip(str)
```

string.strip(" lua + tex = luatex ")

```
lua + tex = luatex
```

split splitlines checkedsplit

The line splitter is a special case of the generic splitter. The `split` function can get a string as well an `lpeg` pattern. The `checkedsplit` function removes empty substrings.

```
local t = string.split        (str, pattern)
local t = string.split        (str, lpeg)
local t = string.checkedsplit (str, lpeg)
local t = string.splitlines   (str)
```

string.split("a, b,c, d", ",")

```
t={  "a",  " b",  "c",  " d", }
```

string.split("p.q,r", lpeg.S(",."))

```
t={  "p",  "q",  "r", }
```

string.checkedsplit(";one;;two", ";")

```
t={  "one",  "two", }
```

string.splitlines("lua\ntex nic")

```
t={  "lua",  "tex nic", }
```

quoted unquoted

You will hardly need these functions. The `quoted` function can normally be avoided using the `format` pattern `%q`. The `unquoted` function removes single or double quotes but only when the string starts and ends with the same quote.

```
local q = string.quoted  (str)
local u = string.unquoted(str)
```

string.quoted([[test]])

```
"test"
```

string.quoted([[test"test]])

```
"test\"test"
```

string.unquoted([["test]])

```
"test
```

string.unquoted([["t\"est"]])

```
t\"est
```

string.unquoted([["t\"est"x]])

```
"t\"est"x
```

string.unquoted("\'test\'")

```
test
```

count

The function `count` returns the number of times that a given pattern occurs. Beware: if you want to deal with utf strings, you need the variant that sits in the `lpeg` namespace.

```
local n = count(str,pattern)
```

string.count("test me", "e")

```
2
```

limit

This function can be handy when you need to print messages that can be rather long. By default, three periods are appended when the string is chopped.

```
print(limit(str,max,sentinel)
```

string.limit("too long", 4)

```
 ...
```

string.limit("too long", 4, " (etc)")

```
too lon (etc)
```

is_empty

A string considered empty by this function when its length is zero or when it only contains spaces.

```
if is_empty(str) then ... end
```

string.is_empty("")

```
true
```

string.is_empty(" ")

```
true
```

string.is_empty(" ? ")

```
false
```

escapedpattern topattern

These two functions are rather specialized. They come in handy when you need to escape a pattern, i.e. prefix characters with a special meaning by a `%`.

```
local e = escapedpattern(str, simple)
local p = topattern     (str, lowercase, strict)
```

The simple variant does less escaping (only `-.?*` and is for instance used in wildcard patterns when globbing directories. The `topattern` function always does the simple escape. A strict pattern gets anchored to the beginning and end. If you want to see what these functions do you can best look at their implementation.

8.6 Numbers

This library is under construction and will be replaced when we have the bit library.

8.7 Lpegs

For LuaTEX and ConTEXt MkIV the `lpeg` library came at the right moment as we can use it in lots of places. An in-depth discussion makes no sense as it's easier to look into `l-lpeg.lua`, so we stick to an overview.[2] Most function return an `lpeg` object that can be used in a match. In time critical situations it's more efficient to use the match on a predefined pattern that to create the pattern new each time. Patterns are cached so there is no penalty in predefining a pattern. So, in the following example, the `splitter` that splits at the asterisk will only be created once.

```
local splitter_1 = lpeg.splitat("*")
local splitter_2 = lpeg.splitat("*")

local n, m = lpeg.match(splitter_1,"2*4")
local n, m = lpeg.match(splitter_2,"2*4")
```

[lua] match print P R S V C Cc Cs ...

The `match` function does the real work. Its first argument is a `lpeg` object that is created using the functions with the short uppercase names.

```
local P, R, C, Ct = lpeg.P, lpeg.R, lpeg.C, lpeg.Ct

local pattern = Ct((P("[") * C(R("az")^0) * P(']') + P(1))^0)

local words = lpeg.match(pattern,"a [first] and [second] word")
```

In this example the words between square brackets are collected in a table. There are lots of examples of `lpeg` in the ConTEXt code base.

anywhere

```
local p = anywhere(pattern)
```

[2] If you search the web for `lua lpeg` you will end up at the official documentation and tutorial.

lpeg.match(lpeg.Ct((lpeg.anywhere("->")/"!")^0), "oeps->what->more")

```
t={  "!",  "!", }
```

splitter splitat firstofsplit secondofsplit

The `splitter` function returns a pattern where each match gets an action applied. The action can be a function, table or string.

```
local p = splitter(pattern, action)
```

The `splitat` function returns a pattern that will return the split off parts. Unless the second argument is `true` the splitter keeps splitting

```
local p = splitat(separator,single)
```

When you need to split off a prefix (for instance in a label) you can use:

```
local p = firstofsplit(separator)
local p = secondofsplit(separator)
```

The first function returns the original when there is no match but the second function returns `nil` instead.

lpeg.match(lpeg.Ct(lpeg.splitat("->",false)), "oeps->what->more")

```
t={  "oeps",  "what",  "more", }
```

lpeg.match(lpeg.Ct(lpeg.splitat("->",false)), "oeps")

```
t={  "oeps", }
```

lpeg.match(lpeg.Ct(lpeg.splitat("->",true)), "oeps->what->more")

```
t={  "oeps",  "what->more", }
```

lpeg.match(lpeg.Ct(lpeg.splitat("->",true)), "oeps")

```
t={  "oeps", }
```

lpeg.match(lpeg.firstofsplit(":"), "before:after")

```
before
```

lpeg.match(lpeg.firstofsplit(":"), "whatever")

```
whatever
```

lpeg.match(lpeg.secondofsplit(":"), "before:after")

```
after
```

lpeg.match(lpeg.secondofsplit(":"), "whatever")

```
nil
```

split checkedsplit

The next two functions have counterparts in the `string` namespace. They return a table with the split parts. The second function omits empty parts.

```
local t = split       (separator,str)
local t = checkedsplit(separator,str)
```

lpeg.split(",","a,b,c")

```
t={ "a", "b", "c", }
```

lpeg.split(",",",a,,b,c,")

```
t={ "", "a", "", "b", "c", "", }
```

lpeg.checkedsplit(",",",a,,b,c,")

```
t={ "a", "b", "c", }
```

stripper keeper replacer

These three functions return patterns that manipulate a string. The `replacer` gets a mapping table passed.

```
local p = stripper(str or pattern)
local p = keeper  (str or pattern)
local p = replacer(mapping)
```

lpeg.match(lpeg.stripper(lpeg.R("az")), "[-a-b-c-d-]")

```
[-----]
```

lpeg.match(lpeg.stripper("ab"), "[-a-b-c-d-]")

```
[---c-d-]
```

lpeg.match(lpeg.keeper(lpeg.R("az")), "[-a-b-c-d-]")

```
abcd
```

lpeg.match(lpeg.keeper("ab"), "[-a-b-c-d-]")

```
ab
```

lpeg.match(lpeg.replacer{{"a","p"},{"b","q"}}, "[-a-b-c-d-]")

```
[-p-q-c-d-]
```

balancer

One of the nice things about `lpeg` is that it can handle all kind of balanced input. So, a function is provided that returns a balancer pattern:

```
local p = balancer(left,right)
```

lpeg.match(lpeg.Ct((lpeg.C(lpeg.balancer("{","}"))+1)^0),"{a} {b{c}}")

```
t={  "{a}",  "{b{c}}", }
```

lpeg.match(lpeg.Ct((lpeg.C(lpeg.balancer("((","]"))+1)^0),"((a] ((b((c]]")

```
t={  "((a]",  "((b((c]]", }
```

counter count

The `counter` function returns a function that returns the length of a given string. The `count` function differs from its counterpart living in the `string` namespace in that it deals with utf and accepts strings as well as patterns.

```
local fnc = counter(pattern)
local len = count(str,what)
```

lpeg.count("äáàa","ä")

```
1
```

lpeg.count("äáàa",lpeg.P("á") + lpeg.P("à"))

```
2
```

UP US UR

In order to make working with utf-8 input somewhat more convenient a few helpers are provided.

```
local p = lpeg.UP(utfstring)
local p = lpeg.US(utfstring)
local p = lpeg.UR(utfpair)
local p = lpeg.UR(first,last)
```

lpeg.count("äáàa",lpeg.UP("áà"))

```
1
```

lpeg.count("äáàa",lpeg.US("àá"))

2

lpeg.count("äáàa",lpeg.UR("aá"))

4

lpeg.count("äáàa",lpeg.UR("àá"))

2

lpeg.count("äáàa",lpeg.UR(0x0000,0xFFFF))

4

patterns

The following patterns are available in the `patterns` table in the `lpeg` namespace:

`HEX` `alwaysmatched` `anything` `balanced` `beginline` `beginofstring` `cardinal` `cfloat` `chartonumber` `cnumber` `colon` `comma` `commaspacer` `digit` `dimenpair` `dquote` `emptyline` `endofstring` `eol` `equal` `escaped` `float` `hex` `hexadecimal` `integer` `letter` `linesplitter` `lowercase` `nested` `newline` `nodquote` `nonspacer` `nonwhitespace` `nosquote` `number` `oct` `octal` `period` `semicolon` `sign` `somecontent` `space` `spaceortab` `spacer` `squote` `stripzeros` `tab` `textline` `underscore` `undouble` `unquoted` `unsingle` `unspacer` `uppercase` `urlescaper` `urlsplitter` `utf8` `utf8byte` `utf8char` `utf8four` `utf8one` `utf8three` `utf8two` `utfbom` `utflinesplitter` `utftype` `validutf8` `validutf8char` `whitespace` `xml`

There will probably be more of them in the future.

8.8 IO

The `io` library is extended with a couple of functions as well and variables but first we mention a few predefined functions.

[lua] open popen...

The IO library deals with in- and output from the console and files.

```
local f  = io.open(filename)
```

When the call succeeds `f` is a file object. You close this file with:

```
f:close()
```

Reading from a file is done with `f:read(...)` and writing to a file with `f:write(...)`. In order to write to a file, when opening a second argument has to be given, often `wb` for writing (binary) data. Although there are more efficient ways, you can use the `f:lines()` iterator to process a file line by line.

You can open a process with `io.popen` but dealing with this one depends a bit on the operating system.

fileseparator pathseparator

The value of the following two strings depends on the operating system that is used.

```
io.fileseparator
io.pathseparator
```

io.fileseparator

```
\
```

io.pathseparator

```
;
```

loaddata savedata

These two functions save you some programming. The first function loads a whole file in a string. By default the file is loaded in binary mode, but when the second argument is `true`, some interpretation takes place (for instance line endings). In practice the second argument can best be left alone.

```
io.loaddata(filename,textmode)
```

Saving the data is done with:

```
io.savedata(filename,str)
io.savedata(filename,tab,joiner)
```

When a table is given, you can optionally specify a string that ends up between the elements that make the table.

exists size noflines

These three function don't need much comment.

```
io.exists(filename)
io.size(filename)
io.noflines(fileobject)
io.noflines(filename)
```

characters bytes readnumber readstring

When I wrote the icc profile loader, I needed a few helpers for reading strings of a certain length and numbers of a given width. Both accept five values of `n`: `-4`, `-2`, `1`, `2` and `4` where the negative values swap the characters or bytes.

```
io.characters(f,n) --
io.bytes(f,n)
```

The function `readnumber` accepts five sizes: `1`, `2`, `4`, `8`, `12`. The string function handles any size and strings zero bytes from the string.

```
io.readnumber(f,size)
io.readstring(f,size)
```

Optionally you can give the position where the reading has to start:

```
io.readnumber(f,position,size)
io.readstring(f,position,size)
```

ask

In practice you will probably make your own variant of the following function, but at least a template is there:

```
io.ask(question,default,options)
```

For example:

```
local answer = io.ask("choice", "two", { "one", "two" })
```

8.9 File

The file library is one of the larger core libraries that comes with ConTEXt.

dirname basename extname nameonly

We start with a few filename manipulators.

```
local path   = file.dirname(name,default)
local base   = file.basename(name)
local suffix = file.extname(name,default) -- or file.suffix
local name   = file.nameonly(name)
```

file.dirname("/data/temp/whatever.cld")

```
/data/temp
```

file.dirname("c:/data/temp/whatever.cld")

```
c:/data/temp
```

file.basename("/data/temp/whatever.cld")

```
whatever.cld
```

file.extname("c:/data/temp/whatever.cld")

```
cld
```

file.nameonly("/data/temp/whatever.cld")

```
whatever
```

addsuffix replacesuffix

These functions are used quite often:

```
local filename = file.addsuffix(filename, suffix, criterium)
local filename = file.replacesuffix(filename, suffix)
```

The first one adds a suffix unless one is present. When `criterium` is `true` no checking is done and the suffix is always appended. The second function replaces the current suffix or add one when there is none.

file.addsuffix("whatever","cld")

```
whatever.cld
```

file.addsuffix("whatever.tex","cld")

```
whatever.tex
```

file.addsuffix("whatever.tex","cld",true)

```
whatever.tex.cld
```

file.replacesuffix("whatever","cld")

```
whatever.cld
```

file.replacesuffix("whatever.tex","cld")

```
whatever.cld
```

is_writable is_readable

These two test the nature of a file:

```
file.is_writable(name)
```

```
file.is_readable(name)
```

splitname join collapsepath

Instead of splitting off individual components you can get them all in one go:

```
local drive, path, base, suffix = file.splitname(name)
```

The `drive` variable is empty on operating systems other than MS Windows. Such components are joined with the function:

```
file.join(...)
```

The given snippets are joined using the `/` as this is rather platform independent. Some checking takes place in order to make sure that nu funny paths result from this. There is also `collapsepath` that does some cleanup on a path with relative components, like `..`.

file.splitname("a:/b/c/d.e")

```
a /b/c/ d e
```

file.join("a","b","c.d")

```
a/b/c.d
```

file.collapsepath("a/b/../c.d")

```
a/c.d
```

file.collapsepath("a/b/../c.d",true)

```
e:/context/manuals/cld-mkiv/a/c.d
```

splitpath joinpath

By default splitting a execution path specification is done using the operating system dependant separator, but you can force one as well:

```
file.splitpath(str,separator)
```

The reverse operation is done with:

```
file.joinpath(tab,separator)
```

Beware: in the following examples the separator is system dependent so the outcome depends on the platform you run on.

file.splitpath("a:b:c")

```
t={  "a:b:c", }
```

file.splitpath("a;b;c")

```
t={ "a", "b", "c", }
```

file.joinpath({"a","b","c"})

```
a;b;c
```

robustname

In workflows filenames with special characters can be a pain so the following function replaces characters other than letters, digits, periods, slashes and hyphens by hyphens.

```
file.robustname(str,strict)
```

file.robustname("We don't like this!")

```
We-don-t-like-this-
```

file.robustname("We don't like this!",true)

```
we-don-t-like-this
```

readdata writedata

These two functions are duplicates of functions with the same name in the `io` library.

copy

There is not much to comment on this one:

```
file.copy(oldname,newname)
```

is_qualified_path is_rootbased_path

A qualified path has at least one directory component while a rootbased path is anchored to the root of a filesystem or drive.

```
file.is_qualified_path(filename)
file.is_rootbased_path(filename)
```

file.is_qualified_path("a")

```
false
```

file.is_qualified_path("a/b")

```
true
```

file.is_rootbased_path("a/b")

```
false
```

file.is_rootbased_path("/a/b")

```
true
```

8.10 Dir

The `dir` library uses functions of the `lfs` library that is linked into LuaTEX.

current

This returns the current directory:

```
dir.current()
```

glob globpattern globfiles

The `glob` function collects files with names that match a given pattern. The pattern can have wildcards: `*` (oen of more characters), `?` (one character) or `**` (one or more directories). You can pass the function a string or a table with strings. Optionally a second argument can be passed, a table that the results are appended to.

```
local files = dir.glob(pattern,target)
local files = dir.glob({pattern,...},target)
```

The target is optional and often you end up with simple calls like:

```
local files = dir.glob("*.tex")
```

There is a more extensive version where you start at a path, and applies an action to each file that matches the pattern. You can either or not force recursion.

```
dir.globpattern(path,patt,recurse,action)
```

The `globfiles` function collects matches in a table that is returned at the end. You can pass an existing table as last argument. The first argument is the starting path, the second arguments controls analyzing directories and the third argument has to be a function that gets a name passed and is supposed to return `true` or `false`. This function determines what gets collected.

```
dir.globfiles(path,recurse,func,files)
```

makedirs

With `makedirs` you can create the given directory. If more than one name is given they are concatinated.

```
dir.makedirs(name,...)
```

expandname

This function tries to resolve the given path, including relative paths.

```
dir.expandname(str)
```

dir.expandname(".")

```
e:/context/manuals/cld-mkiv
```

8.11 URL

split hashed construct

This is a specialized library. You can split an `url` into its components. An url is constructed like this:

```
foo://example.com:2010/alpha/beta?gamma=delta#epsilon
```

scheme	foo://
authority	example.com:2010
path	/alpha/beta
query	gamma=delta
fragment	epsilon

A string is split into a hash table with these keys using the following function:

```
url.hashed(str)
```

or in strings with:

```
url.split(str)
```

The hash variant is more tolerant than the split. In the hash there is also a key `original` that holds the original url and and the boolean `noscheme` indicates if there is a scheme at all.

The reverse operation is done with:

```
url.construct(hash)
```

url.hashed("foo://example.com:2010/alpha/beta?gamma=delta#epsilon")

```
t={ ["authority"]="example.com:2010", ["fragment"]="epsilon", ["noscheme"]=fa
["original"]="foo://example.com:2010/alpha/beta?gamma=delta#epsilon", ["path"]=
["query"]="gamma=delta", ["scheme"]="foo", }
```

url.hashed("alpha/beta")

```
t={  ["authority"]="",  ["fragment"]="",  ["noscheme"]=true,  ["original"]="alph
["path"]="alpha/beta",  ["query"]="",  ["scheme"]="file", }
```

url.split("foo://example.com:2010/alpha/beta?gamma=delta#epsilon")

```
t={  "foo",  "example.com:2010",  "alpha/beta",  "gamma=delta",  "epsilon",
}
```

url.split("alpha/beta")

```
t={  "",  "",  "",  "",  "", }
```

hasscheme addscheme filename query

There are a couple of helpers and their names speaks for themselves:

```
url.hasscheme(str)
url.addscheme(str,scheme)
url.filename(filename)
url.query(str)
```

url.hasscheme("http://www.pragma-ade.com/cow.png")

```
true
```

url.hasscheme("www.pragma-ade.com/cow.png")

```
false
```

url.addscheme("www.pragma-ade.com/cow.png","http://")

```
http://www.pragma-ade.com/cow.png
```

url.addscheme("www.pragma-ade.com/cow.png")

```
file:///www.pragma-ade.com/cow.png
```

url.filename("http://www.pragma-ade.com/cow.png")

```
http://www.pragma-ade.com/cow.png
```

url.query("a=b&c=d")

```
t={  ["a"]="b",  ["c"]="d", }
```

8.12 OS

[lua luatex] env setenv getenv

In ConTEXt normally you will use the resolver functions to deal with the environment and files. However, a more low level interface is still available. You can query and set environment variables with two functions. In addition there is the `env` table as interface to the environment. This threesome replaces the built in functions.

```
os.setenv(key,value)
os.getenv(key)
os.env[key]
```

[lua] execute

There are several functions for running programs. One comes directly from Lua, the otheres come with LuaTEX. All of them are are overloaded in ConTEXt in order to get more control.

```
os.execute(...)
```

[luatex] spawn exec

Two other runners are:

```
os.spawn(...)
os.exec (...)
```

The `exec` variant will transfer control from the current process to the new one and not return to the current job. There is a more detailed explanation in the LuaTEX manual.

resultof launch

The following function runs the command and returns the result as string. Multiple lines are combined.

```
os.resultof(command)
```

The next one launches a file assuming that the operating system knows what application to use.

```
os.launch(str)
```

type name platform libsuffix binsuffix

There are a couple of strings that reflect the current machinery: `type` returns either `windows` or `unix`. The variable `name` is more detailed: `windows`, `msdos`, `linux`, `macosx`, etc. If you also want the architecture you can consult `platform`.

```
local t = os.type
local n = os.name
local p = os.platform
```

These three variables as well as the next two are used internally and normally they are not needed in your applications as most functions that matter are aware of what platform specific things they have to deal with.

```
local s = os.libsuffix
local b = os.binsuffix
```

These are string, not functions.

os.type

```
windows
```

os.name

```
windows
```

os.platform

```
mswin
```

os.libsuffix

```
dll
```

os.binsuffix

```
exe
```

[lua] time

The built in time function returns a number. The accuracy is implementation dependent and not that large.

os.time()

```
1290802571
```

[luatex] times gettimeofday

Although Lua has a built in type os.time function, we normally will use the one provided by LuaTEX as it is more precise:

```
os.gettimeofday()
```

There is also a more extensive variant:

`os.times()`

This one is platform dependent and returns a table with `utime` (use time), `stime` (system time), `cutime` (children user time), and `cstime` (children system time).

os.gettimeofday()

```
1290802571.1707
```

os.times()

```
t={  ["cstime"]=0,  ["cutime"]=0,  ["stime"]=0,  ["utime"]=1290802571.1863,
}
```

runtime

More interesting is:

`os.runtime()`

which returns the time spent in the application so far.

os.runtime()

```
6.0528111457825
```

Sometimes you need to add the timezone to a verbose time and the following function does that for you.

`os.timezone(delta)`

os.timezone()

```
1
```

os.timezone(1)

```
+01:00
```

os.timezone(-1)

```
+01:00
```

uuid

A version 4 UUID can be generated with:

`os.uuid()`

The generator is good enough for our purpose.

os.uuid()

```
ffd54133-4dad-a1b0-0cc7-474d99198d01
```

8.13 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 spent the runtime on it:

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

9 The Lua interface code

9.1 Introduction

There is a lot of Lua code in MkIV. Much is not exposed and a lot of what is exposed is not meant to be used directly at the Lua end. But there is also functionality and data that can be accessed without side effects. This chapter only discussed what makes sense.

In the following sections a subset of the built in functionality is discussed. There are often more functions alongside those presented but they might change or disappear. So, if you use undocumented features, be sure to tag them somehow in your source code so that you can check them out when there is an update. Best would be to have more functionality defined local so that it is sort of hidden but that would be unpractical as for instance functions are often used in other modules and or have to be available at the TEX end.

It might be tempting to add your own functions to namespaces created by ConTEXt or maybe overload some existing ones. Don't do this. First of all, there is no guarantee that your code will not interfere, nor that it overloads future functionality. Just use your own namespace. Also, future versions of ConTEXt might have a couple of protection mechanisms built in. Without doubt the following sections will be extended as soon as interfaces become more stable.

9.2 Characters

There are quite some data tables defined but the largest is the character database. You can consult this table any time you want but you're not supposed to add or change its content. Future versions may carry more information. The table can be accessed using an unicode number. A relative simple entry looks as follows:

characters.data[0x00C1]

```
{ adobename="Aacute", category="lu", contextname="Aacute", description="LATI
CAPITAL LETTER A WITH ACUTE", direction="l", lccode=0x00E1, linebreak="al",
shcode=0x0041, specials={ "char", 0x0041, 0x0301 }, unicodeslot=0x00C1,
}
```

Much of this is rather common information but some of it is specific for use with ConTEXt. Some characters have even more information, for instance those that deal with mathematics:

characters.data[0x2190]

```
{ adobename="arrowleft", category="sm", cjkwd="a", description="LEFTWARDS
ARROW", direction="on", linebreak="ai", mathspec={ { class="relation",
name="leftarrow", }, { class="relation", name="gets", }, {
class="under", name="underleftarrow", }, { class="over", name="over
}, }, mathstretch="h", unicodeslot=0x2190, }
```

Not all characters have a real entry. For instance most cjk characters are virtual and share the same data:

characters.data[0x3456]

```
{  category="lo",  cjkwd="w",  description="<CJK Ideograph Extension A>",
direction="l",  linebreak="id",  range={   first=0x3400,   last=0x4DB5,  },
}
```

You can also access the table using utf characters:

characters.data["ä"]

```
{  ["adobename"]="adieresis",  ["category"]="ll",  ["contextname"]="adiaeresis",
["description"]="LATIN SMALL LETTER A WITH DIAERESIS",  ["direction"]="l",
["linebreak"]="al",  ["shcode"]=97,  ["specials"]={ "char", 97, 776 },  ["uccode
["unicodeslot"]=228, }
```

A more verbose string access is also supported:

characters.data["U+0070"]

```
{  adobename="p",  category="ll",  cjkwd="na",  description="LATIN SMALL
LETTER P",  direction="l",  linebreak="al",  mathclass="variable",  uccode=0x005
unicodeslot=0x0070, }
```

Another (less usefull) table contains information about ranges in this character table. You can access this table using rather verbose names, or you can use collapsed lowercase variants.

characters.blocks["CJK Compatibility Ideographs"]

```
{  0xF900,  0xFAFF,  "CJK Compatibility Ideographs", }
```

characters.blocks["hebrew"]

```
{  0x0590,  0x05FF,  "Hebrew", }
```

characters.blocks["combiningdiacriticalmarks"]

```
{  0x0300,  0x036F,  "Combining Diacritical Marks", }
```

Some fields can be accessed using functions. This can be handy when you need that information for tracing purposes or overviews. There is some overhead in the function call, but you get some extra testing for free. You can use characters as well as numbers as index.

characters.contextname("ä")

```
adiaeresis
```

characters.adobename(228)

```
adieresis
```

characters.description("ä")

```
LATIN SMALL LETTER A WITH DIAERESIS
```

The category is normally a two character tag, but you can also ask for a more verbose variant:

characters.category(228)

```
ll
```

characters.category(228,true)

```
Letter Lowercase
```

The more verbose category tags are available in a table:

characters.categorytags["lu"]

```
Letter Uppercase
```

There are several fields in a character entry that help us to remap a character. The `lccode` indicates the lowercase code point and the `uccode` to the uppercase code point. The `shcode` refers to one or more characters that have a similar shape.

characters.shape ("ä")

```
97
```

characters.uccode("ä")

```
196
```

characters.lccode("ä")

```
ä
```

characters.shape (100)

```
100
```

characters.uccode(100)

```
68
```

characters.lccode(100)

```
100
```

You can use these function or access these fields directly in an entry, but we also provide a few virtual tables that avoid accessing the whole entry. This method is rather efficient.

characters.lccodes["ä"]

```
228
```

characters.uccodes["ä"]

```
196
```

characters.shcodes["ä"]

```
97
```

characters.lcchars["ä"]

```
ä
```

characters.ucchars["ä"]

```
Ä
```

characters.shchars["ä"]

```
a
```

As with other tables, you can use a number instead of an utf character. Watch how we get a table for multiple shape codes but a string for multiple shape characters.

characters.lcchars[0x00C6]

```
æ
```

characters.ucchars[0x00C6]

```
Æ
```

characters.shchars[0x00C6]

```
AE
```

characters.shcodes[0x00C6]

```
{  65,  69, }
```

These codes are used when we manipulate strings. Although there are `upper` and `lower` functions in the `string` namespace, the following ones are the real ones to be used in critical situations.

characters.lower("ÀÁÂÃÄÅàáâãäå")

```
àáâãäåàáâãäå
```

characters.upper("ÀÁÂÃÄÅàáâãäå")

```
ÀÁÂÃÄÅÀÁÂÃÄÅ
```

characters.shaped("ÀÁÂÃÄÅàáâãäå")

```
AAAAAAaaaaaa
```

A rather special one is the following:

characters.lettered("Only 123 letters + count!")

```
Onlyletterscount
```

With the second argument is true, spaces are kept and collapsed. Leading and trailing spaces are stripped.

characters.lettered("Only 123 letters + count!",true)

```
Only letters count
```

Access to tables can happen by number or by string, although there are some limitations when it gets too confusing. Take for instance the number `8` and string `"8"`: if we would interpret the string as number we could never access the entry for the character eight. However, using more verbose hexadecimal strings works okay. The remappers are also available as functions:

characters.tonumber("a")

```
97
```

characters.fromnumber(100)

```
d
```

characters.fromnumber(0x0100)

```
Ā
```

characters.fromnumber("0x0100")

```
Ā
```

characters.fromnumber("U+0100")

```
Ā
```

In addition to the already mentioned category information you can also use a more direct table approach:

characters.categories["ä"]

```
ll
```

characters.categories[100]

```
ll
```

In a similar fashion you can test if a given character is in a specific category. This can save a lot of tests.

characters.is_character[characters.categories[67]]

```
true
```

characters.is_character[67]

```
true
```

characters.is_character[characters.data[67].category]

```
true
```

characters.is_letter[characters.data[67].category]

```
true
```

characters.is_command[characters.data[67].category]

```
nil
```

Another virtual table is the one that provides access to special information, for instance about how a composed character is made up of components.

characters.specialchars["ä"]

```
a
```

characters.specialchars[100]

```
d
```

The outcome is often similar to output that uses the shapecode information.

9.3 Fonts

There is a lot of code that deals with fonts but most is considered to be a black box. When a font is defined, its data is collected and turned into a form that TEX likes. We keep most of that data available at the Lua end so that we can later use it when needed.

A font instance is identified by its id, which is a number where zero is reserved for the so called `nullfont`. The current font id can be requested by the following function.

fonts.currentid()

```
47
```

The `fonts.current()` call returns the table with data related to the current id. You can access the data related to any id as follows:

```
local tfmdata = fonts.identifiers[number]
```

Not all entries in the table make sense for the user as some are just meant to drive the font initialization at the TEX end or the backend. The next table lists the most important ones. Some of the tables are just shortcuts to en entry in one of the `shared` subtables.

`ascender`	number	the height of a line conforming the font
`descender`	number	the depth of a line conforming the font
`italicangle`	number	the angle of the italic shapes (if present)
`designsize`	number	the design size of the font (if known)
`size`	number	the size in scaled points if the font instance
`factor`	number	the multiplication factor for unscaled dimensions
`hfactor`	number	the horizontal multiplication factor
`vfactor`	number	the vertical multiplication factor
`extend`	number	the horizontal scaling to be used by the backend
`slant`	number	the slanting to be applied by the backend
`characters`	table	the scaled character (glyph) information (tfm)
`descriptions`	table	the original unscaled glyph information (otf, afm, tfm)
`indices`	table	the mapping from unicode slot to glyph index
`unicodes`	table	the mapoing from glyph names to unicode
`marks`	table	a hash table with glyphs that are marks as entry
`parameters`	table	the font parameters as TEX likes them
`mathconstants`	table	the OpenType math parameters
`mathparameters`	table	a reference to the `MathConstants` table
`shared`	table	a table with information shared between instances
`unique`	table	a table with information unique for this instance
`unscaled`	table	the unscaled (intermediate) table
`goodies`	table	the ConTEXt specific extra font information
`fonts`	table	the table with references to other fonts
`cidinfo`	table	a table with special information for the backend
`filename`	string	the full path of the loaded font
`fontname`	string	the font name as specified in the font (limited in size)
`fullname`	string	the complete font name as specified in the font
`name`	string	the (short) name of the font
`psname`	string	the (unique) name of the font as used by the backend
`hash`	string	the hash that makes this instance unique
`id`	number	the id (number) that TEX will use for this instance
`type`	string	an idicator if the font is `virtual` or `real`
`format`	string	a qualification for this font, e.g. `opentype`
`mode`	string	the ConTEXt processing mode, `node` or `base`

The `parameters` table contains variables that are used by TEX itself. You can use numbers as index and these are equivalent to the so called `\fontdimen` variables. More convenient is is to access by name:

`slant`	the slant per point (seldom used)
`space`	the interword space

spacestretch	the interword stretch
spaceshrink	the interword shrink
xheight	the x-height (not per se the heigth of an x)
quad	the so called em-width (often the width of an emdash)
extraspace	additional space added in specific situations

The math parameters are rather special and explained in the LuaTEX manual. Quite certainly you never have to touch these parameters at the Lua end.

En entry in the `characters` table describes a character if we have entries within the Unicode range. There can be entries in the private area but these are normally variants of a shape or special math glyphs.

name	the name of the character
index	the index in the raw font table
height	the scaled height of the character
depth	the scaled depth of the character
width	the scaled height of the character
tounicode	a utf-16 string representing the conversion back to unicode
expansion_factor	a multiplication factor for (horizontal) font expansion
left_protruding	a multiplication factor for left side protrusion
right_protruding	a multiplication factor for right side protrusion
italic	the italic correction
next	a pointer to the next character in a math size chain
vert_variants	a pointer to vertical variants conforming OpenType math
horiz_variants	a pointer to horizontal variants conforming OpenType math
top_accent	information with regards to math top accents
mathkern	a table describing stepwise math kerning (following the shape)
kerns	a table with intercharacter kerning dimensions
ligatures	a (nested) table describing ligatures that start with this character
commands	a table with commands that drive the backend code for a virtual shape

Not all entries are present for each character. Also, in so called `node` mode, the `ligatures` and `kerns` tables are empty because in that case they are dealt with at the Lua end and not by TEX.

Say that you run into a glyph node and want to access the data related to that glyph. Given that variable `n` points to the node, the most verbose way of doing that is:

```
local g = fonts.identifiers[n.id].characters[n.char]
```

Given the speed of LuaTEX this is quite fast. Another method is the following:

```
local g = fonts.characters[n.id][n.char]
```

For some applications you might want fast access to critical parameters, like:

```
local quad    = fonts.quads    [n.id][n.char]
local xheight = fonts.xheights[n.id][n.char]
```

but that only makes sense when you don't access more than one such variable at the same time.

Among the shared tables is the feature specification:

fonts.current().shared.features

```
{ ["analyze"]=true, ["features"]=true, ["kern"]=true, ["liga"]=true,
["number"]=1, ["tlig"]=true, ["trep"]=true, }
```

As features are a prominent property of OpenType fonts, there are a few datatables that can be used to get their meaning.

fonts.otf.tables.features['liga']

```
Standard Ligatures
```

fonts.otf.tables.languages['nld']

```
Dutch
```

fonts.otf.tables.scripts['arab']

```
Arabic
```

There is a rather extensive font database built in but discussing its interface does not make much sense. Most usage happens automatically when you use the `name:` and `spec:` methods of defining fonts and the `mtx-fonts` script is built on top of it.

table.sortedkeys(fonts.names.data)

```
{ "cache_uuid", "cache_version", "datastate", "fallbacks", "families",
"files", "mappings", "sorted_fallbacks", "sorted_families", "sorted_mappings",
"specifications", "statistics", "version", }
```

You can load the database (if it's not yet loaded) with:

```
names.load(reload,verbose)
```

When the first argument is true, the database will be rebuild. The second arguments controls verbosity.

Defining a font normally happens at the TeX end but you can also do it in Lua.

```
local id, fontdata = fonts.definers.define {
    lookup = "file",               -- use the filename (file spec name)
    name   = "pagella-regular",    -- in this case the filename
    size   = 10*65535,             -- scaled points
    global = false,                -- define the font globally
    cs     = "MyFont",             -- associate the name \MyFont
    method = "featureset",         -- featureset or virtual (* or @)
```

```
    sub    = nil,                   -- no subfont specifier
    detail = "whatever",            -- the featureset (or whatever method applies)
}
```

In this case the `detail` variable defines what featureset has to be applied. You can define such sets at the Lua end too:

```
fonts.definers.specifiers.presetcontext (
    "whatever",
    "default",
    {
        mode = "node",
        dlig = "yes",
    }
)
```

The first argument is the name of the featureset. The second argument can be an empty string or a reference to an existing featureset that will be taken as starting point. The final argument is the featureset. This can be a table or a string with a comma separated list of key/value pairs.

9.4 Nodes

Nodes are the building blocks that make a document reality. Nodes are linked into lists and at various moments in the typesetting process you can manipulate them. Deep down in ConTEXt we use quite some Lua magic to manipulate lists of nodes. Therefore it is no surprise that we have some tracing available. Take the followingbox.

This box contains characters and glue between the words. The box is already constructed. There can also be kerns between characters, but of course only if the font provides such a feature. Let's inspect this box:

nodes.toutf(tex.box[0])

`It's in all those nodes.`

nodes.toutf(tex.box[0].list)

`It's in all those nodes.`

This tracer returns the text and spacing and recurses into nested lists. The next tracer does not do this and marks non glyph nodes as `[-]`:

nodes.listtoutf(tex.box[0])

`[-]`

nodes.listtoutf(tex.box[0].list)

`It'[-]s[-]in[-][-][-]t[-]hose[-]nodes.`

A more verbose tracer is the next one. It does show a bit more detailed information about the glyphs nodes.

nodes.tosequence(tex.box[0])

```
hlist
```

nodes.tosequence(tex.box[0].list)

```
U+0049:I U+0074:t U+0027:' kern U+0073:s glue U+0069:i U+006E:n glue hlist
glue U+0074:t kern U+0068:h U+006F:o U+0073:s U+0065:e glue U+006E:n U+006F:o
U+0064:d U+0065:e U+0073:s U+002E:.
```

The fourth tracer does not show that detail and collapses sequences of similar node types.

nodes.idstostring(tex.box[0])

```
[hlist]
```

nodes.idstostring(tex.box[0].list)

```
[3*glyph] [kern] [glyph] [glue] [2*glyph] [glue] [hlist] [glue] [glyph] [kern]
[4*glyph] [glue] [6*glyph]
```

The number of nodes in a list is identified with the `count` function. Nested nodes are counted too.

nodes.count(tex.box[0])

```
28
```

nodes.count(tex.box[0].list)

```
27
```

There are functions to check node types and node id's:

```
local str = node.type(1)
local num = node.id("vlist")
```

These are basic LuaTEX functions. In addition to those we also provide a few mapping tables. There are two tables that map node id's to strings and backwards:

`nodes.nodecodes`	regular nodes, some fo them are sort of private to the engine
`nodes.noadcodes`	math nodes that later on are converted into regular nodes

Nodes can have subtypes. Again we have tables that map the subtype numbers onto meaningfull names and reverse.

`nodes.listcodes`	subtypes of `hlist` and `vlist` nodes
`nodes.kerncodes`	subtypes of `kern` nodes
`nodes.gluecodes`	subtypes of `glue` nodes (skips)

`nodes.glyphcodes`	subtypes of `glyph` nodes, the subtype can change
`nodes.mathcodes`	math specific subtypes
`nodes.fillcodes`	these are not really subtypes but indicate the strength of the filler
`nodes.whatsitcodes`	subtypes of a rather large group of extension nodes

Some of the names of types and subtypes have underscores but you can omit them when you use these tables. You can use tables like this as follows:

```
local glyph_code = nodes.nodecodes.glyph
local kern_code  = nodes.nodecodes.kern
local glue_code  = nodes.nodecodes.glue

for n in nodes.traverse(list) do
    local id == n.id
    if id == glyph_code then
        ...
    elseif id == kern_code then
        ...
    elseif id == glue_code then
        ...
    else
        ...
    end
end
```

You only need to use such temporary variables in time critical code. In spite of what you might think, lists are not that long and given the speed of Lua (and successive optimizations in LuaTEX) looping over a paragraphs is rather fast.

Nodes are created using `node.new`. If you study the ConTEXt code you will notice that there are quite some functions in the `nodes.pool` namespace, like:

```
local g = nodes.pool.glyph(fnt,chr)
```

Of course you need to make sure that the font id is valid and that the referred glyph in in the font. You can use the allocators but don't mess with the code in the `pool` namespace as this might interfere with its usage all over ConTEXt.

The `nodes` namespace provides a couple of helpers and some of them are similar to ones provided in the `node` namespace. This has practical as well as historic reasons. For instance some were prototypes functions that were later built in.

```
local head, current      = nodes.before (head, current, new)
local head, current      = nodes.after  (head, current, new)
local head, current      = nodes.delete (head, current)
local head, current      = nodes.replace(head, current, new)
local head, current, old = nodes.remove (head, current)
```

Another category deals with attributes:

```
nodes.setattribute       (head, attribute, value)
nodes.unsetattribute     (head, attribute)
nodes.setunsetattribute  (head, attribute, value)
nodes.setattributes      (head, attribute, value)
nodes.unsetattributes    (head, attribute)
nodes.setunsetattributes(head, attribute, value)
nodes.hasattribute       (head, attribute, value)
```

9.5 Resolvers

All io is handled by functions in the `resolvers` namespace. Most of the code that you find in the `data-*.lua` files is of litle relevance for users, especially at the Lua end, so we won't discuss it here in great detail.

The resolver code is modelled after the kpse library that itself implements the TEX Directory Structure in combination with a configuration file. However, we go a bit beyond this structure, for instance in integrating support for other resources that file systems. We also have our own configuration file. But important is that we still support a similar logic too so that regular configurations are dealt with.

During a run LuaTEX needs files of a different kind: source files, font files, images, etc. In practice you will probably only deal with source files. The most fundamental function is `findfile`. The first argument is the filename to be found. A second optional argument indicates the filetype.

The following table relates so called formats to suffixes and variables in the configuration file.

variable	format	suffix
AFMFONTS	afm	afm
	adobe font metric	
	adobe font metrics	
	bib	bib
	bst	bst
FONTCIDMAPS	cid	cid cidmap
	cid map	
	cid maps	
	cid file	
	cid files	
FONTFEATURES	fea	fea
	font feature	
	font features	
	font feature file	
	font feature files	
TEXFORMATS	fmt	fmt
	format	
	tex format	

FONTCONFIG_PATH	fontconfig	
	fontconfig file	
	fontconfig files	
ICCPROFILES	icc	icc
	icc profile	
	icc profiles	
CLUAINPUTS	lib	dll
LUAINPUTS	lua	lua luc tma tmc
MPMEMS	mem	mem
	metapost format	
MPINPUTS	mp	mp
OFMFONTS	ofm	ofm tfm
	omega font metric	
	omega font metrics	
OPENTYPEFONTS	otf	otf
	opentype	
	opentype font	
	opentype fonts	
OVFFONTS	ovf	ovf vf
	omega virtual font	
	omega virtual fonts	
T1FONTS	pfb	pfb pfa
	type1	
	type 1	
	type1 font	
	type 1 font	
	type1 fonts	
	type 1 fonts	
TEXINPUTS	tex	tex mkiv mkiv mkii
TEXMFSCRIPTS	texmfscript	rb pl py
	texmfscripts	
	script	
	scripts	
TFMFONTS	tfm	tfm
	tex font metric	
	tex font metrics	
TTFONTS	ttf	ttf ttc dfont
	truetype	
	truetype font	
	truetype fonts	
	truetype collection	
	truetype collections	
	truetype dictionary	
	truetype dictionaries	

VFFONTS	vf virtual font virtual fonts	vf

There are a couple of more formats but these are not that relevant in the perspective of ConTEXt.

When a lookup takes place, spaces are ignored and formats are normalized to lowercase.

file.strip(resolvers.findfile("context.tex"),"tex/")

```
c:/data/develop/context/sources/context.tex
```

file.strip(resolvers.findfile("context.mkiv"),"tex/")

```
c:/data/develop/context/sources/context.mkiv
```

file.strip(resolvers.findfile("context"),"tex/")

```
c:/data/develop/context/sources/context.tex
```

file.strip(resolvers.findfile("data-res.lua"),"tex/")

```
c:/data/develop/context/sources/data-res.lua
```

file.strip(resolvers.findfile("lmsans10-bold"),"tex/")

file.strip(resolvers.findfile("lmsans10-bold.otf"),"tex/")

```
texmf/fonts/opentype/public/lm/lmsans10-bold.otf
```

file.strip(resolvers.findfile("lmsans10-bold","otf"),"tex/")

```
texmf/fonts/opentype/public/lm/lmsans10-bold.otf
```

file.strip(resolvers.findfile("lmsans10-bold","opentype"),"tex/")

```
texmf/fonts/opentype/public/lm/lmsans10-bold.otf
```

file.strip(resolvers.findfile("lmsans10-bold","opentypefonts"),"tex/")

```
texmf/fonts/opentype/public/lm/lmsans10-bold.otf
```

file.strip(resolvers.findfile("lmsans10-bold","opentype fonts"),"tex/")

```
texmf/fonts/opentype/public/lm/lmsans10-bold.otf
```

The plural variant of this function returns one or more matches.

resolvers.findfiles("texmfcnf.lua","cnf")

```
{  "c:/data/develop/tex-context/tex/texmf-local/web2c/texmfcnf.lua", }
```

resolvers.findfiles("context.tex","")

```
{  "c:/data/develop/context/sources/context.tex",  "c:/data/develop/tex-context/
}
```

9.6 Mathematics (math)

todo

9.7 Graphics (grph)

todo

9.8 Languages (lang)

todo

9.9 MetaPost (mlib)

todo

9.10 LuaTEX (luat)

todo

9.11 Tracing (trac)

todo

10 Callbacks

10.1 Introduction

The LuaTEX engine provides the usual basic TEX functionality plus a bit more. It is a deliberate choice not to extend the core engine too much. Instead all relevant processes can be overloaded by new functionality written in Lua. In ConTEXt callbacks are wrapped in a protective layer: on the one hand there is extra functionality (usually interfaced through macros) and on the other hand users can pop in their own handlers using hooks. Of course a plugged in function has to do the right thing and not mess up the data structures. In this chapter the layer on top of callbacks is described.

10.2 Actions

Nearly all callbacks in LuaTEX are used in ConTEXt. In the following list the callbacks tagged with `enabled` are used and frozen, the ones tagged `disabled` are blocked and never used, while the ones tagged `undefined` are yet unused.

`buildpage_filter`	`enabled`	vertical spacing etc (mvl)
`char_exists`	`undefined`	
`define_font`	`enabled`	definition of fonts (tfmtable preparation)
`find_data_file`	`enabled`	find file using resolver
`find_enc_file`	`enabled`	find file using resolver
`find_font_file`	`enabled`	find file using resolver
`find_format_file`	`enabled`	find file using resolver
`find_image_file`	`enabled`	find file using resolver
`find_map_file`	`enabled`	find file using resolver
`find_opentype_file`	`enabled`	find file using resolver
`find_output_file`	`enabled`	find file using resolver
`find_pk_file`	`enabled`	find file using resolver
`find_read_file`	`enabled`	find file using resolver
`find_sfd_file`	`enabled`	find file using resolver
`find_truetype_file`	`enabled`	find file using resolver
`find_type1_file`	`enabled`	find file using resolver
`find_vf_file`	`enabled`	find file using resolver
`find_write_file`	`enabled`	find file using resolver
`finish_pdffile`	`enabled`	
`hpack_filter`	`enabled`	all kind of horizontal manipulations
`hyphenate`	`disabled`	normal hyphenation routine, called elsewhere
`kerning`	`disabled`	normal kerning routine, called elsewhere
`ligaturing`	`disabled`	normal ligaturing routine, called elsewhere
`linebreak_filter`	`enabled`	breaking paragraps into lines
`mlist_to_hlist`	`enabled`	preprocessing math list
`open_read_file`	`enabled`	open file for reading

post_linebreak_filter	enabled	all kind of horizontal manipulations (after par break)
pre_dump	enabled	lua related finalizers called before we dump the format
pre_linebreak_filter	enabled	all kind of horizontal manipulations (before par break)
pre_output_filter	undefined	
process_input_buffer	disabled	actions performed when reading data
process_output_buffer	disabled	actions performed when writing data
read_data_file	enabled	read file at once
read_enc_file	enabled	read file at once
read_font_file	enabled	read file at once
read_map_file	enabled	read file at once
read_opentype_file	undefined	read file at once
read_pk_file	enabled	read file at once
read_sfd_file	enabled	read file at once
read_truetype_file	undefined	read file at once
read_type1_file	undefined	read file at once
read_vf_file	enabled	read file at once
show_error_hook	enabled	
start_page_number	enabled	actions performed at the beginning of a shipout
start_run	enabled	actions performed at the beginning of a run
stop_page_number	enabled	actions performed at the end of a shipout
stop_run	enabled	actions performed at the end of a run
token_filter	undefined	
vpack_filter	enabled	vertical spacing etc

Eventually all callbacks will be used so don't rely on undefined callbacks not being protected. Some callbacks are only set when certain functionality is enabled.

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

All callbacks related to file handling, font definition and housekeeping are frozen and cannot be overloaded. A reason for this are that we need some kind of protection against misuse. Another reason is that we operate in a well defined environment, the so called TEX directory structure, and we don't want to mess with that. And of course, the overloading permits ConTEXt to provide extensions beyond regular engine functionality.

So as a fact we only open up some of the node list related callbacks and these are grouped as follows:

category	**callback**	**usage**
processors	pre_linebreak_filter	called just before the paragraph is broken into lines

	hpack_filter	called just before a horizontal box is constructed
finalizers	post_linebreak_filter	called just after the paragraph has been broken into lines
shipouts	no callback yet	applied to the box (or xform) that is to be shipped out
mvlbuilders	buildpage_filter	called after some material has been added to the main vertical list
vboxbuilders	vpack_filter	called when some material is added to a vertical box
math	mlist_to_hlist	called just after the math list is created, before it is turned into an horizontal list

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

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

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

As you can see, in this case the function gets no head passed (at least not currently). This example also assumes that you know how to access the right items. The arguments and return values are given below.[3]

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

10.3 Tasks

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

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

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

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

category	function
`before`	unset
`normalizers`	`fonts.collections.process` `fonts.checkers.missing`

`characters`	`typesetters.directions.handler`
	`typesetters.cases.handler`
	`typesetters.breakpoints.handler`
	`scripts.preprocess`

`words`	`builders.kernel.hyphenation` `languages.words.check`

`fonts`	`builders.paragraphs.solutions.splitters.split`
	`nodes.handlers.characters`
	`nodes.injections.handler`
	`nodes.handlers.protectglyphs`
	`builders.kernel.ligaturing`
	`builders.kernel.kerning`
	`nodes.handlers.stripping`

`lists`	`typesetters.spacings.handler`
	`typesetters.kerns.handler`
	`typesetters.digits.handler`

`after`	unset

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

The finalizer tasks are:

category	function
`before`	unset
`normalizers`	unset
`fonts`	`builders.paragraphs.solutions.splitters.optimize`
`lists`	`nodes.handlers.graphicvadjust`
`after`	unset

Shipouts concern:

category	function
`before`	unset

`normalizers`	`nodes.handlers.cleanuppage`
	`nodes.references.handler`
	`nodes.destinations.handler`
	`nodes.rules.handler`
	`nodes.shifts.handler`
	`structures.tags.handler`
	`nodes.handlers.accessibility`
	`nodes.handlers.backgrounds`

`finishers`	`attributes.colors.handler`
	`attributes.transparencies.handler`
	`attributes.colorintents.handler`
	`attributes.negatives.handler`
	`attributes.effects.handler`
	`attributes.viewerlayers.handler`

`after`	unset

There are not that many mvlbuilder tasks currently:

category	**function**
`before`	unset

`normalizers`	`streams.collect`
	`nodes.handlers.migrate`
	`builders.vspacing.pagehandler`

`after`	unset

The vboxbuilder perform similar tasks:

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

In the future we expect to have more parbuilder tasks. Here again there are subtasks that depend on the current typesetting environment, so this is the right spot for language specific treatments.

The following actions are applied just before the list is passed on the the output routine. The return value is a vlist.

Both the parbuilders and pagebuilder tasks are unofficial and not yet meant for users.

Finally, we have tasks related to the math list:

category	**function**
`before`	unset

normalizers	noads.handlers.relocate noads.handlers.collapse noads.handlers.resize noads.handlers.respace noads.handlers.check noads.handlers.tags
builders	builders.kernel.mlist_to_hlist
after	unset

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

You can create your own task list with:

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

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

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

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

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

Functions can also be removed:

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

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

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

The same can be done with a complete category:

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

There is one function left:

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

This function returns a function that when called will perform the tasks. In this case the function takes two extra arguments in addition to `head`.[4]

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

10.4 Paragraph and page builders

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

10.5 Some examples

todo

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

11 Backend code

11.1 Introduction

In ConTEXt we've always separated the backend code in so called driver files. This means that in the code related to typesetting only calls to the api take place, and no backend specific code is to be used. Currently a pdf backend is supported as well as an xml export.[5]

Some ConTEXt users like to add their own pdf specific code to their styles or modules. However, such extensions can interfere with existing code, especially when resources are involved. Therefore the construction of pdf data structures and resources is rather controlled and has to be done via the official helper macros.

11.2 Structure

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

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

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

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

11.3 Data types

There are several datatypes in pdf and we support all of them one way or the other.

[5] This chapter is derived from an article on these matters. You can find nore information in `hybrid.pdf`.

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

While writing additional backend code, we mostly create dictionaries.

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

In this case the indirect object can look like:

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

The LuaTEX manual mentions primitives like `\pdfobj`, `\pdfannot`, `\pdfcatalog`, etc. However, in MkIV no such primitives are used. You can still use many of them but those that push data into document or page related resources are overloaded to do nothing at all.

In the Lua backend code you will find function calls like:

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

Equaly valid is:

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

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

```
local r = lpdf.immediateobject(tostring(d))
```

or using the wrapper (which permits tracing):

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

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

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

An example of using references is:

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

We have the following creators. Their arguments are optional.

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

tostring(lpdf.null())

```
null
```

tostring(lpdf.number(123))

```
123
```

tostring(lpdf.constant("whatever"))

```
/whatever
```

tostring(lpdf.string("just a string"))

```
(just a string)
```

tostring(lpdf.unicode("just a string"))

```
<feff006a007500730074002000610020007300740072069006e0067>
```

tostring(lpdf.boolean(true))

```
true
```

tostring(lpdf.array { 1, lpdf.constant("c"), true, "str" })

```
[ 1 /c true (str) ]
```

tostring(lpdf.dictionary { a=1, b=lpdf.constant("c"), d=true, e="str" })

```
<< /a 1 /d true /e (str) /b /c >>
```

tostring(lpdf.reference(123))

```
123 0 R
```

tostring(lpdf.verbose("whatever"))

```
whatever
```

11.4 Managing objects

Flushing objects is done with:

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

Reserving object is or course possible and done with:

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

Such an object is flushed with:

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

We also support named objects:

```
lpdf.reserveobject("myobject")
```

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

A delayed object is created with:

```
local ref = pdf.delayedobject(data)
```

The data will be flushed later using the object number that is returned (`ref`). When you expect that many object with the same content are used, you can use:

```
local obj = lpdf.shareobject(data)
local ref = lpdf.shareobjectreference(data)
```

This one flushes the object and returns the object number. Already defined objects are reused. In addition to this code driven optimization, some other optimization and reuse takes place but all that happens without user intervention. Only use this when it's really needed as it might consume more memory and needs more processing time.

11.5 Resources

While LuaTeX itself will embed all resources related to regular typesetting, MkIV has to take care of embedding those related to special tricks, like annotations, spot colors, layers, shades, transparencies, metadata, etc. Because third party modules (like tikz) also can add resources we provide some macros that makes sure that no interference takes place:

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

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

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

One is free to use the Lua interface instead, as there one has more possibilities but when code is shared with other macro packages the macro interface makes more sense. The names of the Lua functions are similar, like:

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

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

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

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

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

11.6 Annotations

You can use the Lua functions that relate to annotations etc. but normally you will use the regular ConTEXt user interface. You can look into some of the `lpdf-*` modules to see how special annotations can be dealt with.

11.7 Tracing

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

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

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

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

The reference related backend mechanisms have their own trackers. When you write code that generates pdf, it also helps to look in the pdf file so see if things are done right. In that case you need to disable compression:

```
\nopdfcompression
```

12 XML

12.1 Introduction

Being a popular input format, xml deserves some attention, especially because ConTEXt has a parser built-in. The parser is written in Lua and therefore interfacing is quite convenient. Of course you can skip this chapter if you don't run into xml at all or when the regular TEX interface to xml is enough for your jobs.

13 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.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.

14 Special commands

There are a few functions in the `context` namespace that are no macros at the TEX end.

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

Another useful command is:

```
context.settracing(true)
```

There are a few tracing options that you can set at the TEX end:

```
\enabletrackers[context.files]
\enabletrackers[context.trace]
```

A few macros have special functions at the Lua end. One of them is `\char`. The function makes sure that the characters ends up right. The same is true for `\chardef`. So, you don't need to mess around with `\relax` or trailing spaces as you would do at the TEX end in order to tell the scanner to stop looking ahead.

```
context.char(123)
```

Other examples of macros that have optimized functions are `\par`, `\bgroup` and `\egroup`.

Index

needs checking, incomplete