Solutions for the CTAN quiz (on p. 18)

1. `https://ctan.org/topics/cloud`
2. `https://ctan.org/lion`
3. `https://ctan.org/credits`
4. `https://ctan.org/help/supported-browsers`
5. `https://ctan.org/mirrors`
6. `https://ctan.org/mirrors/register`
7. `https://ctan.org/pkg`
8. `https://ctan.org/author`
9. `https://ctan.org/ctan-ann`
10. `https://ctan.org/search?ext=new`
11. `https://ctan.org/upload`
12. `https://ctan.org/help/submit`
13. `https://ctan.org/user/settings`
14. `https://ctan.org/author/knuth`
15. `https://ctan.org/help/json`
16. `https://ctan.org/tex-archive`
17. `https://ctan.org/home`
18. `https://ctan.org/guestbook`
19. `https://ctan.org/lugs`
20. `https://ctan.org/incoming`

prefixation
Dist. by Universal Uclick
© John Atkinson, Wrong Hands
Here they come again. These guys always try to start something.
re
un
do
©John Atkinson, Wrong Hands • gocomics.com/wrong-hands • wronghands1.com

Comic by John Atkinson (`http://wronghands1.com`).

From Lua 5.2 to 5.3

Hans Hagen

When we started with LuaTEX we used Lua 5.1 and moved to 5.2 when that became available. We didn't run into issues then because there were no fundamental changes that could not be dealt with. However, when Lua 5.3 was announced in 2015 we were not sure if we should make the move. The main reason was that we'd chosen Lua because of its clean design which meant that we had only one number type: double. In 5.3 on the other hand, deep down a number can be either an integer or a floating point quantity.

Internally TEX is mostly (up to) 32-bit integers and when we go from Lua to TEX we round numbers. Nonetheless one can expect some benefits in using integers. Performance-wise we didn't expect much, and memory consumption would be the same too. So, the main question then was: can we get the same output and not run into trouble due to possible differences in serializing numbers; after all TEX is about stability. The serialization aspect is for instance important when we compare quantities and/or use numbers in hashes.

Apart from this change in number model, which comes with a few extra helpers, another extension in 5.3 was that bit-wise operations are now part of the language. The lpeg library is still not part of stock Lua. There is some minimal UTF8 support, but less than we provide in LuaTEX already. So, looking at these changes, we were not in a hurry to update. Also, it made sense to wait till this important number-related change was stable.

But, a few years later, we still had it on our agenda to test, and after the ConTEXt 2017 meeting we decided to give it a try; here are some observations. A quick test was just dropping in the new Lua code and seeing if we could make a ConTEXt format. Indeed that was no big deal but a test run failed because at some point a (for instance) `1` became a `1.0`. It turned out that serializing has some side effects. And with some ad hoc prints for tracing (in the LuaTEX source) I could figure out what went on. How numbers are seen can (to some extent) be deduced from the `string.format` function, which is in Lua a combination of parsing, splitting and concatenation combined with piping to the C code `sprintf` function.[1]

[1] Actually, at some point I decided to write my own formatter on top of `format` and I ended up with splitting as well. It's only now that I realize why this is working out so well (in terms of performance): simple format (single items) are passed more or less directly to `sprintf` and as Lua itself is

code	number	fmt	out	type
`local a =  2   * (1/2) print(string.format("%s",  a),math.type(x))`	2 * (1/2)	`s`	1.0	`float`
`local b =  2   * (1/2) print(string.format("%d",  b),math.type(x))`	2 * (1/2)	`d`	1	`float`
`local c =  2           print(string.format("%d",  c),math.type(x))`	2	`d`	2	`integer`
`local d = -2           print(string.format("%d",  d),math.type(x))`	−2	`d`	2	`integer`
`local e =  2   * (1/2) print(string.format("%i",  e),math.type(x))`	2 * (1/2)	`i`	1	`float`
`local f =  2.1         print(string.format("%.0f",f),math.type(x))`	2.1	`.0f`	2	`float`
`local g =  2.0         print(string.format("%.0f",g),math.type(x))`	2.0	`.0f`	2	`float`
`local h =  2.1         print(string.format("%G",  h),math.type(x))`	2.1	`G`	2.1	`float`
`local i =  2.0         print(string.format("%G",  i),math.type(x))`	2.0	`G`	2	`float`
`local j =  2           print(string.format("%.0f",j),math.type(x))`	2	`.0f`	2	`integer`
`local k = -2           print(string.format("%.0f",k),math.type(x))`	−2	`.0f`	2	`integer`

Figure 1: Various number representation in Lua 5.3: code at left, summary and output at right.

Figure 1 gives many examples, demonstrating that we have to be careful when we need these numbers represented as strings. In ConTEXt the number of places where we had to check for that was not that large; in fact, only some hashing related to font sizes had to be done using explicit rounding.

Another surprising side effect is the following. Instead of:

```
local n = 2^6
```

we now need to use:

```
local n = 0x40
```

or just:

```
local n = 64
```

because we don't want this to be serialized to `64.0` which is due to the fact that a power results in a float. One can wonder if this makes sense when we apply it to an integer.

At any rate, once we could process a file, two documents were chosen for a performance test. Some experiments with loops and casts had demonstrated that we could expect a small performance hit and indeed, this was the case. Processing the LuaTEX manual takes 10.7 seconds with 5.2 on my 5-year-old laptop and 11.6 seconds with 5.3. If we consider that ConTEXt spends 50% of its time in Lua, then we see a 20% performance penalty. Processing the Metafun manual (which has lots of MetaPost images) went from less than 20 seconds (LuaJITTEX does it in 16 seconds) up to more than 27 seconds. So there we lose more than 50% on the Lua end. When we observed these kinds of differences, Luigi and I immediately got into debugging mode, partly out of curiosity, but also because consistent performance is important to us.

fast, due to some caching, the overhead is small compared to the built-in splitter method. And the ConTEXt formatter has many more options and is extensible.

Because these numbers made no sense, we traced different sub-mechanisms and eventually it became clear that the reason for the speed penalty was that the core `string.format` function was behaving quite badly in the `mingw` cross-compiled binary, as seen by this test:

```
local t = os.clock()
for i=1,1000*1000 do
 -- local a = string.format("%.3f",1.23)
 -- local b = string.format("%i",123)
    local c = string.format("%s",123)
end
print(os.clock()-t)
```

	lua 5.3	lua 5.2	texlua 5.3	texlua 5.2
`a`	0.43	0.54	3.71 (0.47)	0.53
`b`	0.18	0.24	3.78 (0.17)	0.22
`c`	0.26	0.68	3.67 (0.29)	0.66

The 5.2 binaries perform the same but the 5.3 Lua binary greatly outperforms LuaTEX, and so we had to figure out why. After all, all this integer optimization could bring some gain! It took us a while to figure this out. The numbers in parentheses are the results after fixing this.

Because font internals are specified in integers one would expect a gain in running:

```
mtxrun --script font --reload force
```

and indeed that is the case. On my machine a scan results in 2561 registered fonts from 4906 read files and with 5.2 that takes 9.1 seconds while 5.3 needs a bit less: 8.6 seconds (with the bad format performance) and even less once that was fixed. For a test:

```
\setupbodyfont[modern]     \tf \bf \it \bs
\setupbodyfont[pagella]    \tf \bf \it \bs
\setupbodyfont[dejavu]     \tf \bf \it \bs
\setupbodyfont[termes]     \tf \bf \it \bs
\setupbodyfont[cambria]    \tf \bf \it \bs
\starttext \stoptext
```

This code needs 30% more runtime so the question is: how often do we call `string.format` there? A first run (when we wipe the font cache) needs some 715,000 calls while successive runs need 115,000 calls so that slow down definitely comes from the bad handling of `string.format`. When we drop in a Lua update or whatever other dependency we don't want this kind of impact. In fact, when one uses external libraries that are or can be compiled under the TEX Live infrastructure and the impact would be such, it's bad advertising, especially when one considers the occasional complaint about LuaTEX being slower than other engines.

The good news is that eventually Luigi was able to nail down this issue and we got a binary that performed well. It looks like Lua 5.3.4 (cross)compiles badly with GCC 5.3.0 and 6.3.0.

So in the end caching the fonts takes:

	caching	running
5.2 stock	8.3	1.2
5.3 bugged	12.6	2.1
5.3 fixed	6.3	1.0

So indeed it looks like 5.3 is able to speed up LuaTEX a bit, given that one integrates it in the right way! Using a recent compiler is needed too, although one can wonder when a bad case will show up again. One can also wonder why such a slow down can mostly go unnoticed, because for sure LuaTEX is not the only compiled program.

The next examples are some edge cases that show you need to be aware that 1) an integer has its limits, 2) that hexadecimal numbers are integers and 3) that Lua and LuaJIT can be different in details.

	`print(0xFFFFFFFFFFFFFFFF)`
lua 5.2	1.844674407371e+019
luajit	1.844674407371e+19
lua 5.3	−1

	`print(0x7FFFFFFFFFFFFFFF)`
lua 5.2	9.2233720368548e+018
luajit	9.2233720368548e+18
lua 5.3	9223372036854775807

So, to summarize the process. A quick test was relatively easy: move 5.3 into the code base, adapt a little bit of internals (there were some LuaTEX interfacing bits where explicit rounding was needed), run tests and eventually fix some issues related to the Makefile (compatibility) and C code obscurities (the slow `sprintf`). Adapting ConTEXt was also not much work, and the test suite uncovered some nasty side effects. For instance, the valid 5.2 solution:

```
local s = string.format("02X",u/1024)
local s = string.char        (u/1024)
```

now has to become (both 5.2 and 5.3):

```
local s = string.format("02X",math.floor(u/1024))
local s = string.char        (math.floor(u/1024))
```

or (both 5.2 and (emulated or real) 5.3):

```
local s = string.format("02X",bit32.rshift(u,10))
local s = string.char        (bit32.rshift(u,10))
```

or (only 5.3):

```
local s = string.format("02X",u >> 10))
local s = string.char        (u >> 10)
```

or (only 5.3):

```
local s = string.format("02X",u//1024)
local s = string.char        (u//1024)
```

A conditional section like:

```
if LUAVERSION >= 5.3 then
  local s = string.format("02X",u >> 10))
  local s = string.char        (u >> 10)
else
  local s = string.format("02X",
                         bit32.rshift(u,10))
  local s = string.char  (bit32.rshift(u,10))
end
```

will fail because (of course) the 5.2 parser doesn't like that. In ConTEXt we have some experimental solutions for that but that is beyond this summary.

In the process a few UTF helpers were added to the string library so that we have a common set for LuaJIT and Lua (the `utf8` library that was added to 5.3 is not that important for LuaTEX). For now we keep the `bit32` library on board. Of course we'll not mention all the details here.

When we consider a gain in speed of 5-10% with 5.3 that also means that the gain of LuaJITTEX compared to 5.2 becomes less. For instance in font processing both engines now perform closer to the same.

As I write this, we've just entered 2018 and after a few months of testing LuaTEX with Lua 5.3 we're confident that we can move the code to the experimental branch. This means that we will use this version in the ConTEXt distribution and likely will ship this version as 1.10 in 2019, where it becomes the default. The 2018 version of TEX Live will have 1.07 with Lua 5.2 while intermediate versions of the Lua 5.3 binary will end up on the ConTEXt garden, probably with number 1.08 and 1.09 (who knows what else we will add or change in the meantime).

◇ Hans Hagen
Pragma ADE
`http://pragma-ade.com`