# LUA ENHANCED TEX

# Contents

# Introduction

I started thinking about a Lua extended version of TEX when I played with Lua in the Scite editor (spell checking and such). After some emailing Hartmut Henkel took the challenge and within a few days had the first version of such an integration running. After some experiments, the time was there to involve Taco Hoekwater and interfacing to TEX's internals started taking shape. It will take some time for the interface to mature, and we should take that time. The `\lua` primitive is just a start, because in the end we want to interface to TEX's internals.

Part of the game is to provide a plug–in interface so that we can talk with the (more or less independent) paragraph and page builders that Karl Skoupý is experimenting with and showed at TEX conferences.

The project started as a branch of pdfTEX but in the end it will end up in the main version of Hàn Thế Thành's the original. Given the small footprint of Lua, users who don't use it will not suffer from it.

This manual is a kind of diary, where I keep track of new things and ideas. In the process I will provide:

- a basic macro set for (plain) TEX
- an extended macro set for ConTEXt
- general purpose Lua libraries

This manual is dedicated to Hartmut, Taco, Thanh and Karl, my personal heros of extending and reimplementing TEX who are never tired to try out new things, no matter what weird things I come up with.

Hans Hagen
Pragma ADE
Hasselt NL
August 2005

# 1 Definitions

Before we start playing with Lua, we define a few auxiliary macros. They permits us to write a bit more readable code.

```
\long\def\luatex#1{\scantokens\expandafter{\lua{#1}}}
\long\def\rescan#1{\expanded{\scantokens{#1}}}
```

The `\scantokens` is needed when we want to process the result under the current catcode regime. We will see examples of that later.

The `\lua` command passes its argument and control to the (pending) Lua thread. When executing the code has finished, the result is inserted into the input stream.

```
I wonder for how long we can use \lua { tex.print("Hello World") }
as an example, since it assumes that there is a world left.
```

This piece of source code results in: I wonder for how long we can use Hello World as an example, since it assumes that there is a world left. .

The `tex` table —tables are one of Lua's core mechanisms— contains a predefined `print` function and that one can be used to feed back results into the tex input stream.

In practice there will be more involved than just printing. Lua is pretty well capable of providing TEX some powerful calculation features.

```
\lua{a = 1.5 ; b = 1.8 ; c = a*b ; tex.print(c) ;}

\lua {
    a = 1.5
    b = 1.8
    c = a*b
    tex.print(c)
}
```

The semicolons are only needed when there can be confusion about where a statement ends. Even in the first line of this example, no semicolons are needed, but they make the source a bit more readable.

Those who know ConTEXt may wonder if we need some way of storing Lua code in macros with the purpose of recalling them later on, as we do with METAPOST code. There is no real need for such a mechanism because Lua has functions and Lua itself can collect them for us.

```
\lua {
    function mine.sum(a,b)
        tex.print(a+b)
    end
}
```

Also, because we have an interface to TEX's registers and hash table, we can ask for any information we want. There is no need to explicitly pass for instance page numbers to Lua when entering the page builder, as we do with METAPOST.

# 2 Expansion

You may have noticed by now that the `\lua` command is fully expandable. Only things that you want to end up in the result will end up there: no unexpanded bits and pieces, `\relax`'s etc.

The result is a string comparable with the result of the `\meaning` command: characters with catcodes that identity them as letters and nothing else. So, a `$` is just a dollar, and does not switch to mathmode.

```
\lua        {tex.print("$\string\\sqrt{2} = " .. math.sqrt(2) .. "$")}  \blank
\rescan{\lua{tex.print("$\string\\sqrt{2} = " .. math.sqrt(2) .. "$")}} \blank
\luatex     {tex.print("$\string\\sqrt{2} = " .. math.sqrt(2) .. "$")}  \blank
```

These lines demonstrate that we need to rescan the result in order to let TEX associate the right category codes and so recognize where to start and end math mode.

$\sqrt{2} = 1.4142135623731$

$\sqrt{2} = 1.4142135623731$

$\sqrt{2} = 1.4142135623731$

The `..` operator in this example concatenates strings. Because we want the string `\sqrt` in the result, we duplicate the backslash and prefix it by `\string`. We could redefine `\\` just before we invoke `\lua`:

```
\def\\{\string\\}
```

but this would spoil its fully expandable character so for the moment we will let this rest.

The result of the square root calculation is rather precise. Donald Knuth wrote TEX in such a way that calculations were platform independent. In Lua, calculations are done using floating point arithmics, using standardised protocols. In practice one will not notice this, since the accuracy is more than enough for everyday integer calculations.

# 3 Scripting

Let's present a more complex example. It is no secret that TEX has no real capabilities to manipulate strings. This is no surprise because it's main purpose is to typeset documents and other programs can provide the manipulated input. But what if you want to do it all (or as much as possible) from within TEX?

```
\lua {
    match = {} ;
    match.expression = "" ;
    match.string = "" ;
    match.result = {} ;
    match.start = 0 ;
    match.length = 0 ;
}

\def\matchstring#1#2{\lua {
    match.expression = "#1" ;
    match.string = "#2" ;
    match.result = {} ;
    match.start, match.length,
        match.result[1], match.result[2], match.result[3],
        match.result[4], match.result[5], match.result[6],
        match.result[7], match.result[8], match.result[9] = string.find(match.string,match.expression) ;
} }

\def\matchresult#1{\lua {
    tex.print(match.result[#1]) ;
} }
```

The first call to `\lua` initializes some variables. Next we define two macros, the first one matches an expression with a string, while the second one gives access to the result. However, this implementation is not that efficient because each time that we call `\matchstring`, we pass a lot of code to lua. The next variant is cleaner:

```
\lua {
    match = {} ;
    match.expression = "" ;
    match.string = "" ;
    match.result = {} ;
    match.start = 0 ;
    match.length = 0 ;
}

\lua {
    function match.find(expr, str)
        match.expression = expr ;
        match.string = str ;
        match.result = {} ;
        match.start, match.length,
            match.result[1], match.result[2], match.result[3],
            match.result[4], match.result[5], match.result[6],
            match.result[7], match.result[8], match.result[9] = string.find(match.string,match.expression) ;
    end
}

\def\matchstring#1#2{\lua { match.find("#1","#2") }}
\def\matchresult  #1{\lua { tex.print(match.result[#1]) }}
```

We can use these macros as follows. The `\letterpercent` inserts an percent symbol, which is Lua's escape character in regular expressions.

```
\matchstring
  {(\letterpercent d+) (\letterpercent d+) (\letterpercent d+)}
  {2005 08 08}

\starttabulate[|l|c|r|]
\NC year  \NC \matchresult{1} \NC \number \matchresult{1} \NC \NR
\NC month \NC \matchresult{2} \NC \number \matchresult{2} \NC \NR
\NC day   \NC \matchresult{3} \NC \number \matchresult{3} \NC \NR
\stoptabulate
```

| | | |
|---|---|---|
| year | 2005 | 2005 |
| month | 08 | 8 |
| day | 08 | 8 |

In this example the `\number` command strips the preceding zeros from the result. However, in case of a faulty date, where result returns no number, we run into troubles. Ok, we could add an redundant zero after the `\number` command, but the next alternative is more in sync with what regular expressions can do.

```
\matchstring
  {0*(\letterpercent d+) 0*(\letterpercent d+) 0*(\letterpercent d+)}
  {2005 08 08}
```

```
\starttabulate[|l|c|]
\NC year  \NC \matchresult{1} \NC \NR
\NC month \NC \matchresult{2} \NC \NR
\NC day   \NC \matchresult{3} \NC \NR
\stoptabulate
```

| | |
|---|---|
| year | 2005 |
| month | 8 |
| day | 8 |

# 4 Talking to TEX

An extension like this gains in power when it can interface to TEX's internals. Another benefit is that it provides alternative ways to store and access information. The next example demonstrates how we can use Lua to calculate with TEX dimensions. First we define a 'namespace' table and within that another table.

*The following examples use a preliminary interface to TEX's data structures. This may change.*

```
\lua {
    document = { }
    document.widths = { }
}
```

Next we calculate the widths of characters in the current font and store them in the table.

```
\dostepwiserecurse{`a}{`z}{1} {
    \setbox0\hbox{\char\recurselevel}
    \lua { document.widths[\recurselevel] = tex.wd[0] }
}
```

We calculate the average width and store that value in a variable in the `document` table.

```
\lua {
    local total, n = 0, 0
    for d in pairs(document.widths) do
        total, n = total + document.widths[d], n + 1
    end
    if n>0 then
        document.mean = total/n
    else
        document.mean = 0
    end
}
```

We can access dimension registers in a similar way as the dimensions of boxes, as demonstrated a few lines before. Next we can do things with the calculated mean value.

```
\mathematics {
    \lua { tex.dimen[0] = document.mean } \withoutpt \the\dimen0  =
    \lua { tex.print(document.mean/65536) } \approx
    \lua { tex.print(math.ceil(document.mean/65536)) }
}
```

5.43271 = 5.4327234121469 ≈ 6

We need to keep into mind that only TEX knows how to deal with things meant for TEX: Lua does not know how to handle control sequences and special directives in the input stream. Take the following example.

```
\toks0 = {interesting}

\lua {
    tex.toks[0] = string.gsub(tex.toks[0], "(.)", " (\letterpercent1) ")
}

\the\toks0
```

Here we store a string in a token register. Next we split this string in characters and put parenthesis around it.

(i) (n) (t) (e) (r) (e) (s) (t) (i) (n) (g)

This goes well for simple strings but imagine that we have the string

```
\toks0 = {math f\"ur $\all$}
```

This gives:

(m) (a) (t) (h) ( ) (f) (\) (") (u) (r) ( ) ($) (\) (a) (l) (l) ( ) ($)

and not (m) (a) (t) (h) ( ) (f) (ü) (r) (∀). Apart from the fact that lua sees characters and is not aware of the magic combinations `\"u` and `$\forall$` it just returns letters. If we want to manipulate strings in a more clever way, we need to have access to the internal lists of already typeset material. But, given some constraints, we can do nice things anyway. In the next example we use a quote from Hermann Zapf.

```
\toks0 = {
    Coming back to the use of typefaces in electronic publishing: many of the
    new typographers receive their knowledge and information about the rules
    of typography from books, from computer magazines or the instruction
    manuals which they get with the purchase of a PC or software. There is
    not so much basic instruction, as of now, as there was in the old days,
    showing the differences between good and bad typographic design. Many people
    are just fascinated by their PC's tricks, and think that a widely||praised
    program, called up on the screen, will make everything automatic from now on.
}

\lua {
    str = tex.toks[0]
    str = string.gsub(str, "\letterpercent w+",
        function(w)
            if str.len(w) > 4 then
                return "\string\\color[red]{" .. w .. "}"
            else
                return "\string\\color[green]{" .. w .. "}"
            end
        end
    )
    tex.toks[0] = str
}

\scantokens\expandafter{\the\toks0}
```

The compound word marker || is left untouched and just treated as normal characters.

Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC's tricks, and think that a widely–praised program, called up on the screen, will make everything automatic from now on.

Here is another example of storing and retrieving. This solution is slower that storing the assignments in a token list registers and expanding that one each time when needed, but when assignments take place seldom, it could be an alternative.

```
\lua {
    lccodes = { }
}

\dorecurse{255} {
    \lua {
        lccodes[\recurselevel] = \number\lccode\recurselevel ;
    }
}

\luatex {
    for i in pairs(lccodes) do
        tex.print("\string\\lccode" .. i .. "=" .. lccodes[i] .. " ")
    end
}
```

In this case, the result is just a series of assignments and no text.

In the next example we return to the kind of example code that we started with: fully expandable definitions.

```
\lua {
    interface = {}
    interface.noftests = 0
    function interface.oneoftwo(result)
        interface.noftests = interface.noftests + 1
        if result then
            tex.print("firstoftwoarguments")
        else
            tex.print("secondoftwoarguments")
        end
    end
}
```

This function returns a string, which one depends on the boolean (condition) fed into it. The two possible strings reflect macro names (ConTEXt has a bunch of those):

```
\long\def\firstoftwoarguments #1#2{#1}
\long\def\secondoftwoarguments#1#2{#2}
```

Teh following lines show us what happens:

```
\lua { interface.oneoftwo(true)} \& \lua { interface.oneoftwo(false)}
```

we get: firstoftwoarguments & secondoftwoarguments

Now imagine the following definitions:

```
\def\DoIfElse#1#2%
  {\csname\lua{interface.oneoftwo("#1"=="#2")}\endcsname}
```

We feed the string comparison into the function, which returns a string, which in turn is expanded into a control sequence. Without any change of interference we also keep track of the number of tests. We can ask for this number with:

```
\def\NOfTests
  {\lua{tex.print(interface.noftests)}}
```

The following code shows how to use the test. They all return (typeset) `OK`.

```
\def\xxx{yyy} \def\yyy{yyy}
```

```
\DoIfElse{one}{one}{OK}{NO}
\DoIfElse{two}{one}{NO}{OK}
\DoIfElse\xxx \yyy {OK}{NO}
```

This gives:

OK OK OK

What goes in gets expanded, but when it enters Lua it has been turned into a harmless sequence of characters:

```
\DoIfElse
  {this is a \hbox to \hsize{real} mess}
  {and \rm this is even worse}
  {NO}{OK}
```

This test can be used in an `\edef` and fed back into Lua if needed.

```
\edef\zzz{this \DoIfElse{a}{b}{or}{and} that}

\lua{tex.print("\zzz")}
\lua{tex.print("this \DoIfElse{a}{a}{or}{and} that")}
```

These examples demonstrate that we can comfortably mix TEX and Lua and use either of them for what suits them best. Also keep in mind that nesting Lua calls is no problem either.

```
\lua{tex.print("this \lua{tex.print("---")} that")}
```

Of course we need additional trickery. For instance we need a way to escape characters before they enter Lua, for instance the following will fail:

```
\DoIfElse{one "two" three}{one 'two' three}{OK}{NO}
```

Instead we need something:

```
\DoIfElse
  {\luaesc{one "two" three}}
  {\luaesc{one 'two' three}}
  {OK}{NO}
```

This will comfortably escape the sensitive characters. So, there is some work left to do.