» The Monthy Pythons, were they TEX users, could have written the chickenize macro.«
Paul Isambert

chickenize

Arno Trautmann
arno.trautmann@gmx.de

July 22, 2011

Abstract

This is the package `chickenize`. It allows you to substitute or change the contents of a LuaLATEX document[1], but is actually only for fun. Please *never* use any of the functionality of this package for a production document. The following table informs you shortly about your possibilities and provides links to the Lua functions. A LATEX and plainTEX interface is also offered, see below.

function	effect
chickenize	replaces every word with “chicken”
colorstretch	shows grey boxes that depict the badness of a line
leetspeak	translates every letter into the corresponding 1337 letter
randomuclc	changes randomly between uppercase and lowercase
randomfonts	changes the font randomly between every letter
randomchars	randomizes the whole input
uppercasecolor	adds a color to every uppercase letter

If you have any suggestions or comments, just drop me a mail, I'll be happy to get any response!

Contents

[1] The code is based on pure LuaTEX features, so don't try to use it with any other TEX flavour.

1 How It Works

We make use of LuaTEXs callbacks, especially the `pre_linebreak_filter` and the `post_linebreak_filter`. Hooking a function into these, we can nearly arbitrarily change the contents of the document. If the changes should be on the input-side (replacing with `chicken`), one can use the `pre_linebreak_filter`. Hower, changes like inserting color are best made after the linebreak is finalized, so `post_linebreak_filter` is used for such things.

2 How You Can Use It

There are several ways to make use of this package. As all features are implemented on the Lua side, you can use these functions on their own. If you do so, please consult the corresponding subsections in the ec:implementation] part, because there are some variables that can be adapted to your need.

You can use the following code inside a `\directlua` statement or in a `luacode` environment (or the corresponding thing in your format):

```
luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize the input",1)
```

Replace `"pre` by `"post` to register into the post linebreak filter. The second argument gives the function name; find a list of available functions below. You can give a label as you like in the third argument, and the last argument gives the order in which the functions in the callback are used. If you have no fancy stuff going on, you can safely use `1`.

If you don't want to mess with the Lua side (but please, try it, you'll learn much!), there is a LATEX- as well as a plainTEX interface described in the next section. The commands may not always be on the latest code base – if anything does not work as expected, please tell me and I'll correct it.

2.1 Commands

You have a number of commands at your hand, each of which does some manipulation of the input or output. In fact, the code is easy and straightforward, but be careful, especially when combining things. Apply features step by step so your brain won't be damaged …

Some commands have optional arguments that are *only* available for LaTeX. plainTeX users are mostly capable of finding out how to change things themselfs, but if you are willing to wrap up the code for optional argument processing, don't hesitate sharing it with me ;)

chickenize Replaces every word of the input with the word "chicken". Maybe sometime the replaced word can be changed, but up to now, it's only chicken. To be a bit less static, about every 10^{th} chicken is uppercase. However, the beginning of a sentence is not recognized automatically.[2]

uppercasecolor Makes every uppercase character in the input colored. At the moment, the color is randomized over the full rgb scale, but that will be adjustable once options are well implemented.

randomuclc Changes every character of the input into its uppercase or lowercase variant. Well, guess what the "random" means …

randomfonts Changes the font randomly for every character. If no parameters are given, all fonts that have been loaded are used, especially including math fonts.

leetspeak Translates the input into 1337 speak. If you don't understand that, lern it, n00b.

colorstretch Inspired by Paul Isambert's code, this command prints boxes instead of lines. The greyness of the first (left-hand) box corresponds to the badness of the line, i. e. it is a measure for how much the space between words has been extended to get proper paragraph justification. The second box on the right-hand side shows the amount of stretching/shrinking when font expansion is used. Together the box greyness give you information about how well the overall greyness of the typeset page is. You may specifiy the optional arguments `[(no)keeptext]` to display the text or delete it, also `[(no)colorexpansion]` controls wether or not the font expansion should be evaluated or not.

This functionality is actually the only really usefull implementation of this package …

2.2 text-Versions

The functions of this package might be much more useful if applied only to a short sequence of words or single words instead of the whole document or paragraph. Therefore, most of the above-mentioned commands have a `\text`-version

[2] If you have a nice implementation idea, I'd love to include this!

that takes an argument. `\textrandomcolor{foo}` results in a colored `foo` while the rest of the document keeps its color. However, to achieve this effect, still the whole node list has to be traversed, so it may slow down your document, even if you use `\textrandomcolor` only once. Fortunately, the effect is very small and mostly negligible.[3]

Please don't fool around by mixing a `\text`-version with the non-`\text`-version. If you feel like and are not please with the result, it is up to *you* to provide a stable and working solution.

Part I
Implementation

3 TEX file

```
\input{luatexbase.sty}
\directlua{dofile("chickenize.lua")}

\def\chickenize{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize the input
}
\def\colorstretch{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",colorstretch,"show stretch and
}
\def\leetspeak{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",leet,"transform input to 1337",
}
\def\randomcolor{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"random color",1)}
}
\def\randomfonts{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomfonts,"random fonts",1)}
}
\def\randomuclc{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",randomuclc,"randomize uc/lc char
}
\def\uppercasecolor{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",uppercasecolor,"color all uc ch
}

\newluatexattribute\randcolorattr
\def\textrandomcolor#1{%
\randomcolor%
```

[3] On a 500 pages text-only LaTeX document the dilation is on the order of 10% with `textrandomcolor`, but other manipulations can take much more time. However, you are not supposed to make such long documents with `chickenize`!

```
\setluatexattribute\randcolorattr{42}#1%
\unsetluatexattribute\randcolorattr%
\directlua{randomcolor_onlytext=true}%
\gdef\textrandomcolor#1{%
\setluatexattribute\randcolorattr{42}#1%
\unsetluatexattribute\randcolorattr}
} %% to turn off automatic all-colorizing

```

4 Preparation

Loading of packages and defition of constants. Will change somewhat when migrating to expl3 (?)

```
\input{chickenize}
\RequirePackage{
  expl3,
  xkeyval,
  xparse
}
%% So far, no keys are defined. This will change …
\ExplSyntaxOn
\NewDocumentCommand\chickenizesetup{m}{
  \directlua{#1}
}
```

5 Definition of User-Level Macros

```
\DeclareDocumentCommand\chickenize{}{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize the input
  %% We want to "chickenize" figures, too. So …
  \DeclareDocumentCommand\includegraphics{O{}m}{
     \fbox{Chicken}  %% actually, I'd love to draw a mp graph showing a chicken …
  }
}
%% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
%%           (most probable only available for \LaTeX)

\ExplSyntaxOff  %% because of the : in the domain …
\NewDocumentCommand\balmerpeak{G{}O{-4cm}}{
  \begin{tikzpicture}
  \hspace*{#2}  %% anyhow necessary to fix centering … strange :(
  \begin{axis}
  [width=10cm,height=7cm,
   xmin=-0.005,xmax=0.28,ymin=-0.05,ymax=1,
   xtick={0,0.02,...,0.27},ytick=\empty,
   /pgf/number format/precision=3,/pgf/number format/fixed,
   tick label style={font=\small},
   label style = {font=\Large},
```

```
  xlabel = \fontspec{Punk Nova} BLOOD ALCOHOL CONCENTRATION (\%),
  ylabel = \fontspec{Punk Nova} \rotatebox{-90}{\parbox{3cm}{\center programming\\ skills}}]
   \addplot
     [domain=-0.01:0.27,color=red,samples=250]
     {0.8*exp(-0.5*((x-0.1335)^2)/.00002)+
      0.5*exp(-0.5*((x+0.015)^2)/0.01)
     };
 \end{axis}
 \end{tikzpicture}
}
\ExplSyntaxOn
```

6 Lua Module

This file contains all the necessary functions, sorted alphabetically, not by sense.

First, we set up some constants. These are made global so the code can be manipulated on document level, too.

```
Hhead = node.id("hhead")
RULE = node.id("rule")
GLUE = node.id("glue")
WHAT = node.id("whatsit")
COL = node.subtype("pdf_colorstack")
GLYPH = node.id("glyph")
```

Now we set up the nodes used for all color things. The nodes are whatsits of subtype `pdf_colorstack`.

```
color_push = node.new(WHAT,COL)
color_pop = node.new(WHAT,COL)
color_push.stack = 0
color_pop.stack = 0
color_push.cmd = 1
color_pop.cmd = 2

```

6.1 chickenize

The infamous `\chickenize` macro. Substitutes every word of the input with the given string. This can be elaborated arbitrarily, and whenever I feel like, I might add functionality. So far, only the string replaces the word, and even hyphenation is not possible.

```
chickenstring = "Chicken"

local tbl = font.getfont(font.current())
local space = tbl.parameters.space
local shrink = tbl.parameters.space_shrink
local stretch = tbl.parameters.space_stretch
local match = unicode.utf8.match

function chickenize(head)
```

```
for i in node.traverse_id(37,head) do  --find start of a word
  while ((i.next.id == 37) or (i.next.id == 11) or (i.next.id == 7) or (i.next.id == 0)) do
    i.next = i.next.next
  end

  chicken = {}
  chicken[0] = node.new(37,1)  -- only a dummy for the loop
  for i = 1,string.len(chickenstring) do
    chicken[i] = node.new(37,1)
    chicken[i].font = font.current()
    chicken[i-1].next = chicken[i]
  end

  j = 1
  for s in string.utfvalues(chickenstring) do
  local char = unicode.utf8.char(s)
    chicken[j].char = s
    if match(char,"%s") then
      chicken[j] = node.new(10)
      chicken[j].spec = node.new(47)
      chicken[j].spec.width = space
      chicken[j].spec.shrink = shrink
      chicken[j].spec.stretch = stretch
    end
    j = j+1
  end

  node.insert_before(head,i,chicken[1])
  chicken[1].next = chicken[2] -- seems to be necessary … to be fixed
  chicken[string.len(chickenstring)].next = i.next
end

return head
end
```

6.2 leet

The `leettable` is the substitution scheme. Just add items if you feel to. Maybe we will differ between a light-weight version and a hardcore 1337.

```
leettable = {
  [101] = 51, -- e
  [105] = 49, -- i
  [108] = 49, -- l
  [111] = 48, -- o
  [115] = 53, -- s
  [116] = 55, -- t

  [101-32] = 51, -- e
  [105-32] = 49, -- i
```

```
  [108-32] = 49, -- l
  [111-32] = 48, -- o
  [115-32] = 53, -- s
  [116-32] = 55, -- t
}
```

And the function. So simple that I will not write any

```
function leet(head)
  for line in node.traverse_id(Hhead,head) do
    for i in node.traverse_id(GLYPH,line.head) do
      if leettable[i.char] then
        i.char = leettable[i.char]
      end
    end
  end
  return head
end
```

6.3 randomfonts

Traverses the output and substitutes fonts randomly. A check is done so that the font number is existing. One day, the fonts should be easily given explicitely in terms of \bf etc.

```
randomfontslower = 1
randomfontsupper = 0
%
function randomfonts(head)
  if (randomfontsupper > 0) then rfub = randomfontsupper else rfub = font.max() end  -- either
  for line in node.traverse_id(Hhead,head) do
    for i in node.traverse_id(GLYPH,line.head) do
      i.font = math.random(randomfontslower,rfub)
    end
  end
  return head
end
```

6.4 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```
uclcratio = 0.5 -- so, this can even be changed!
randomuclc = function(head)
  for i in node.traverse_id(37,head) do
    if math.random() < uclcratio then
      i.char = tex.uccode[i.char]
    else
      i.char = tex.lccode[i.char]
end
  end
  return head
```

```
end
```

6.5 randomchars

```
randomchars = function (head)
  for line in node.traverse_id(Hhead,head) do
    for i in node.traverse_id(GLYPH,line.head) do
      i.char = math.floor(math.random()*512)
    end
  end
  return head
end
```

6.6 randomcolor

Setup of the boolean for grey/color, and boundaries for the colors. rgb space is fully used, but greyscale is only used in a visible range, i. e. to 90% instead of 100% white.

```
randomcolor_grey = false
randomcolor_onlytext = false --switch between local and global colorization
-- false means "color everything"
Rgb_lower = 1
rGb_lower = 1
rgB_lower = 1
Rgb_upper = 254
rGb_upper = 254
rgB_upper = 254
grey_lower = 0
grey_upper = 900
```

This function produces the string needed for the pdf color stack. We need values 0]..[1 for the colors.

```
function randomcolorstring()
  if randomcolorgrey then
    return (0.001*math.random(grey_lower,grey_upper)).." g"
  else
    Rgb = math.random(Rgb_lower,Rgb_upper)/255
    rGb = math.random(rGb_lower,rGb_upper)/255
    rgB = math.random(rgB_lower,rgB_upper)/255
    return Rgb..rGb..rgB.." rg"
  end
end
```

The function that does all the colorizing action. It goes through the whole paragraph and looks at every glyph. If the boolean `randomcolor_onlytext` is set, only glyphs with the set attribute will be colored. Elsewise, all glyphs are taken.

```
function randomcolor(head)
  for line in node.traverse_id(0,head) do
    for i in node.traverse_id(37,line.head) do
      if not(randomcolor_onlytext) or (node.has_attribute(i,luatexbase.attributes.randcolorattr
```

```
        color_push.data = randomcolorstring()  -- color or grey string
        line.head = node.insert_before(line.head,i,node.copy(color_push))
        node.insert_after(line.head,i,node.copy(color_pop))
      end
    end
  end
  return head
end
```

6.7 uppercasecolor

Loop through all the nodes and checking whether it is uppercase. If so (and also for small caps), color it.

```
uppercasecolor = function (head)
  for line in node.traverse_id(Hhead,head) do
    for upper in node.traverse_id(GLYPH,line.head) do
      if (((upper.char > 64) and (upper.char < 91)) or
          ((upper.char > 57424) and (upper.char < 57451)))  then  -- for small caps! nice
        color_push.data = randomcolorstring()  -- color or grey string
        line.head = node.insert_before(line.head,upper,node.copy(color_push))
        node.insert_after(line.head,upper,node.copy(color_pop))
      end
    end
  end
  return head
end
```

6.7.1 colorstretch

This function displays the amount of stretching that has been done for each line of an arbitrary document. A well-typeset document should be equally grey over all lines, which is not always possible.

The function shows in fact two boxes: The first (left) box shows the badness, i. e. the amount of stretching the spaces between words. Too much space results in ligth gray, whereas a too dense line is indicated by a dark grey box.

The second box is only usefull if microtypographic extensions are used, e. g. with the `microtype` package under LaTeX. The box color then corresponds to the amount of font expansion in the line. This can be greatly used to show the positive effect of font expansion on the badness of a line!

The base structure of the following code is written by Paul Isambert. Thanks for the code and support, Paul!

Two booleans, `keeptext`, and `colorexpansion`, are used to control the behaviour of the function.

```
keeptext = true
colorexpansion = true
```

After setting the constants, the function starts. It receives the vertical list of the typeset paragraph as `head`, and loops through all horizontal lists.

If font expansion should be shown (`colorexpansion == true`), then the first glyph node is determined and its width compared with the width of the unexpanded glyph. This gives a measure for the expansion factor and is translated into a grey scale.

```
colorstretch = function (head)

 local f = font.getfont(font.current()).characters
 for line in node.traverse_id(Hhead,head) do
   local rule_bad = node.new(RULE)

if colorexpansion then  -- if also the font expansion should be shown
     local g = line.head
       while not(g.id == 37) do
        g = g.next
       end
     exp_factor = g.width / f[g.char].width
     exp_color = .5 + (1-exp_factor)*10 .. " g"
     rule_bad.width = 0.5*line.width  -- we need two rules on each line!
   else
     rule_bad.width = line.width  -- only the space expansion should be shown, only one rule
   end
```

Height and depth of the rules are adapted to print a closed grey pattern, so no white interspace is left.

The glue order and sign can be obtained directly and are translated into a grey scale.

```
   rule_bad.height = tex.baselineskip.width*4/5  -- this should give a quite nice output!
   rule_bad.depth = tex.baselineskip.width*1/5

   local glue_ratio = 0
   if line.glue_order == 0 then
     if line.glue_sign == 1 then
       glue_ratio = .5 * math.min(line.glue_set,1)
     else
       glue_ratio = -.5 * math.min(line.glue_set,1)
     end
   end
   color_push.data = .5 + glue_ratio .. " g"
```

Now, we throw everything together in a way that works. Somehow …

```
-- set up output
   local p = line.head

 -- a rule to immitate kerning all the way back
   local kern_back = node.new(RULE)
   kern_back.width = -line.width

 -- if the text should still be displayed, the color and box nodes are inserted additionally
 -- and the head is set to the color node
   if keeptext then
```

```
279       line.head = node.insert_before(line.head,line.head,node.copy(color_push)) -- make the col
280     else
281       node.flush_list(p)
282       line.head = node.copy(color_push)
283     end
284     node.insert_after(line.head,line.head,rule_bad)  -- then the rule
285     node.insert_after(line.head,line.head.next,node.copy(color_pop)) -- and then pop!
286     tmpnode =  node.insert_after(line.head,line.head.next.next,kern_back)
287
288     -- then a rule with the expansion color
289     if colorexpansion then  -- if also the stretch/shrink of letters should be shown
290       color_push.data = exp_color
291       node.insert_after(line.head,tmpnode,node.copy(color_push))
292       node.insert_after(line.head,tmpnode.next,node.copy(rule_bad))
293       node.insert_after(line.head,tmpnode.next.next,node.copy(color_pop))
294     end
295   end
296   return head
297 end
```

And that's it :)

7 Known Bugs

There are surely some bugs …

???

8 To Dos

Some things that should be implemented but aren't so far or are very poor at the moment:

?