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

chickenize

Arno Trautmann
arno.trautmann@gmx.de

July 25, 2011

Abstract

This is the package `chickenize`. It allows you to substitute or change the contents of a LuaTEX document,[1] but is actually just for fun. Please *never* use any of the functionality of this package for a production document. The following table informs you shortly about some of your possibilities and provides links to the Lua functions. The TEX interface is presented below.

function/command	effect
chickenize	replaces every word with "chicken"
colorstretch	shows grey boxes that depict the badness of a line
leetspeak	translates the (latin-based) input into 1337 5p34k
randomuclc	changes randomly between uppercase and lowercase
randomfonts	changes the font randomly between every letter
randomchars	randomizes the whole input
randomcolor	prints every letter in a random color
uppercasecolor	makes every uppercase letter colored

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 even try to use it with any other TEX flavour. The package is tested under LuaLATEX, and should be working fine with plainLuaTEX. If you tried it with ConTEXt, please share your experience!

Part I
User Documentation

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 – you can either stay on the TEX side or use the Lua functions directly. In fact, the TEX macros are simple wrappers around the functions.

2.1 TEX Commands – Document Wide

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 …

The effect of the commands can be influenced, not with arguments, but only via the `\chickenizesetup` described below.

`\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 10th 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.

`\randomcolor` Does what it's name says.

`\rainbowcolor` Instead of random colors, this command causes the text color to change slowly according to the colors of a rainbow. Do not mix this with `randomcolor`, as it doesn't make any sense.

`\nyanize` A synonym for `rainbowcolor`.

`\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.

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[3] a `\text`-version 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.[4]

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.

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

[3]If they don't have, I did miss that, sorry. Please inform me about such cases.

[4]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`!

2.3 Lua functions

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 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")
```

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`.

3 How to Adjust It

There are several ways to change the behaviour of `chickenize` and its macros. Most of the options are Lua variables and can be set using `\chickenizesetup`.[5] But be *careful!* The argument of `\chickenizesetup` is parsed directly to Lua, therefore you are *not* using a comma-separated key-value list, but uncorrelated Lua commands. The argument must have the syntax `{randomfontslower = 1 randomfontsupper = 0}` instead of `{randomfontslower = 1, randomfontsupper = 0}`. Alright?

However, `\chickenizesetup` is a macro on the TeX side meaning that you can use *both* `%` *or* `--` as comment string.

The following list tries to keep kind of track of the options and variables. There is no guarantee for this list, and if you find something that is missing or doesn't work as described here, please inform me!

`randomfontslower`, `randomfontsupper` = `<int>` These two integer variables determine the span of fonts used for the font randomization. Just play with them a bit to find out what they are doing.

`chickenstring` = `<string>` The string that is printed when using `\chickenize`. So far, this does not really work, especially breaking into lines and hyphenation. Remember that this is Lua input, so a string must be given with quotation marks: `chickenstring = "foo bar"`.

`leettable` = `<table>` From this table, the substitution for 1337 is taken. If you want to add or change an entry, you have to provide the unicode numbers of the characters, e. g. `leettable[101] = 50` replaces every `e` (`101`) with the number `3` (`50`).

`uclcratio` = `<float>` `0.5` Gives the fraction of uppercases to lowercases in the `\randomuclc` mode. A higher number (up to 1) gives more uppercase letters. Guess what a lower number does.

`randomcolor_grey` = `<bool>` `false` For a printer-friendly version, this offers a grey scale instead of an rgb value for `\randomcolor`.

[5] To be honest, this is just `\def`d to `\directlua`. One small advantage of this is that TeX comments do work.

`Rgb_lower`, `rGb_upper` = `<int>` To specify the color space that is used for `\randomcolor`, you can specify six values, the upper and lower value for each color. The uppercase letter in the variable denotes the color, so `rGb_upper` gives the upper value for green etc. Possible values are between `1` and `254`. If you enter anything outside this, your pdf will become invalid and break. For grey scale, use `grey_lower` and `grey_upper`, with values between `0` (black) and `1000` (white), included. Default is `0` to `900` to prevent white letters.

`rainbow_step` = `<float>0.005` This indicates the relative change of color using the rainbow functionality. A value of `1` changes the color in one step from red to yellow, while a value of `0.005` – you get it, I gues. Useful values are below `0.05`, but it depends on the amount of text. The longer the text and the lower the `step`, the nicer the rainbow will be.

`keeptext` = `<bool>` **`false`** This is for the `\colorstretch` command. If set to `true`, the text of your document will be kept. This way, it is easier to identify bad lines and the reason for the badness.

`colorexpansion` = `<bool>` **`true`** If `true`, two bars are shown of which the second one denotes the font expansion. Only usefull if font expansion is used. (You *do* use font expansion, do you?)

Part II
Implementation

4 TeX file

```
\input{luatexbase.sty}
% read the Lua code first
\directlua{dofile("chickenize.lua")}
% then define the global macros. These affect the whole document and will stay active until the functions wi
\def\chickenize{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize")
    luatexbase.add_to_callback("start_page_number",function() texio.write("["..status.total_pages) end ,"csta
    luatexbase.add_to_callback("stop_page_number",function() texio.write(" chickens]") end,"cstoppage")}}  %
\def\unchickenize{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","chickenize")
    luatexbase.remove_from_callback("start_page_number","cstarttpage")
    luatexbase.remove_from_callback("stop_page_number","cstoppage")}}

\def\colorstretch{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",colorstretch,"stretch_expansion")}}
\def\uncolorstretch{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","colorstretch")}}

\def\leetspeak{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",leet,"1337")}}
\def\unleetspeak{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","1337")}}

```

```
\def\rainbowcolor{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"rainbowcolor")
             rainbowcolor = true}}
\def\unrainbowcolor{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","rainbowcolor")
             rainbowcolor = false}}
\let\nyanize\rainbowcolor

\def\randomcolor{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"randomcolor")}}
\def\unrandomcolor{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomcolor")}}

\def\randomfonts{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomfonts,"randomfonts")}}
\def\unrandomfonts{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomfonts")}}

\def\randomuclc{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",randomuclc,"randomuclc")}}
\def\unrandomuclc{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","randomuclc")}}

\def\uppercasecolor{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",uppercasecolor,"uppercasecolor")}}
\def\unuppercasecolor{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","uppercasecolor")}}
```

Now the setup for the `\text`-versions. We utilize LuaTEXs attributes to mark all nodes that should be manipulated. The macros should be `\long` to allow arbitrary input.

```
\newluatexattribute\leetattr
\newluatexattribute\randcolorattr
\newluatexattribute\randfontsattr

\long\def\textleetspeak#1%
  {\setluatexattribute\leetattr{42}#1\unsetluatexattribute\leetattr}
\long\def\textrandomcolor#1%
  {\setluatexattribute\randcolorattr{42}#1\unsetluatexattribute\randcolorattr}
\long\def\textrandomfonts#1%
  {\setluatexattribute\randfontsattr{42}#1\unsetluatexattribute\randfontsattr}
\long\def\textrandomfonts#1%
  {\setluatexattribute\randfontsattr{42}#1\unsetluatexattribute\randfontsattr}
```

Finally, a macro to control the setup. For now, it's only a wrapper for `\directlua`, but it is nice to have a separate abstraction macro. Maybe this will allow for some flexibility.

```
\def\chickenizesetup#1{\directlua{#1}}
```

5 LATEX package

I have decided to keep the LATEX-part of this package as small as possible. So far, it does … nothing usefull, but it provides a `chickenize.sty` that loads `chickenize.tex`. Some code might be implemented to manipulate figures for full chickenization.

```
\input{chickenize}
\RequirePackage{
  expl3,
  xkeyval,
  xparse
}
```

5.1 Definition of User-Level Macros

```
  %% 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/.

\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

chickenize = function(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  --find end of
      i.next = i.next.next
    end

    chicken = {}  -- constructing the node list. Should be done only once?
    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.

```
leet_onlytext = false
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 here the function itself. So simple that I will not write any

```
leet = function(head)
  for line in node.traverse_id(Hhead,head) do
    for i in node.traverse_id(GLYPH,line.head) do
      if not(leetspeak_onlytext) or
         node.has_attribute(i,luatexbase.attributes.leetattr)
      then
        if leettable[i.char] then
          i.char = leettable[i.char]
        end
      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
%
randomfonts = function(head)
  if (randomfontsupper > 0) then  -- fixme: this should be done only once, no? Or at every paragraph?
    rfub = randomfontsupper  -- user-specified value
  else
    rfub = font.max()        -- or just take all fonts
  end
  for line in node.traverse_id(Hhead,head) do
    for i in node.traverse_id(GLYPH,line.head) do
      if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattr) then
        i.font = math.random(randomfontslower,rfub)
      end
    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 or rainbowcolor, 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
rainbowcolor = false

grey_lower = 0
grey_upper = 900

Rgb_lower = 1
rGb_lower = 1
rgB_lower = 1
Rgb_upper = 254
rGb_upper = 254
rgB_upper = 254
```

Variables for the rainbow. 1/rainbow_step*5 is the number of letters used for one cycle, the color changes from red to yellow to green to blue to purple.

```
rainbow_step = 0.005
rainbow_Rgb = 1-step -- we start in the red phase
rainbow_rGb = step   -- values x must always be 0 < x < 1
rainbow_rgB = step
rainind = 1          -- 1:red,2:yellow,3:green,4:blue,5:purple
```

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

```
randomcolorstring = function()
  if randomcolor_grey then
    return (0.001*math.random(grey_lower,grey_upper)).." g"
  elseif rainbowcolor then
    if rainind == 1 then -- red
      rainbow_rGb = rainbow_rGb + rainbow_step
      if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
    elseif rainind == 2 then -- yellow
      rainbow_Rgb = rainbow_Rgb - rainbow_step
      if rainbow_Rgb <= rainbow_step then rainind = 3 end
    elseif rainind == 3 then -- green
      rainbow_rgB = rainbow_rgB + rainbow_step
      rainbow_rGb = rainbow_rGb - rainbow_step
      if rainbow_rGb <= rainbow_step then rainind = 4 end
    elseif rainind == 4 then -- blue
      rainbow_Rgb = rainbow_Rgb + rainbow_step
      if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
    else -- purple
      rainbow_rgB = rainbow_rgB - rainbow_step
      if rainbow_rgB <= rainbow_step then rainind = 1 end
    end
    return rainbow_Rgb..rainbow_rGb..rainbow_rgB.." rg"
  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.

```
randomcolor = function(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))
      then
        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
```

6.8 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 prints two boxes, in fact: 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 better 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
      line.head = node.insert_before(line.head,line.head,node.copy(color_push))
    else
      node.flush_list(p)
      line.head = node.copy(color_push)
    end
    node.insert_after(line.head,line.head,rule_bad)  -- then the rule
    node.insert_after(line.head,line.head.next,node.copy(color_pop)) -- and then pop!
    tmpnode =  node.insert_after(line.head,line.head.next.next,kern_back)

    -- then a rule with the expansion color
    if colorexpansion then  -- if also the stretch/shrink of letters should be shown
      color_push.data = exp_color
      node.insert_after(line.head,tmpnode,node.copy(color_push))
      node.insert_after(line.head,tmpnode.next,node.copy(rule_bad))
      node.insert_after(line.head,tmpnode.next.next,node.copy(color_pop))
    end
  end
  return head
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:

?