

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 LuaT_EX 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 T_EX 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
randomucl	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

I	User Documentation	2
1	How It Works	2
2	How You Can Use It	2
2.1	T _E X Commands – Document Wide	2
2.2	How to Deactivate It	3
2.3	\text-Versions	4
2.4	Lua functions	4
3	How to Adjust It	4

¹The code is based on pure LuaT_EX features, so don't even try to use it with any other T_EX flavour. The package is tested under Lua_{La}T_EX, and should be working fine with plainLuaT_EX. If you tried it with ConT_EXt, please share your experience!

II	Implementation	5
4	T_EX file	5
5	L^AT_EX package	7
5.1	Definition of User-Level Macros	7
6	Lua Module	8
6.1	chickenize	8
6.2	leet	9
6.3	randomfonts	10
6.4	randomucl	10
6.5	randomchars	11
6.6	randomcolor	11
6.7	uppercasecolor	12
6.8	colorstretch	13
7	Known Bugs	15
8	To Dos	15

Part I

User Documentation

1 How It Works

We make use of LuaT_EXs 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`. However, 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 T_EX side or use the Lua functions directly. In fact, the T_EX macros are simple wrappers around the functions.

2.1 T_EX 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 that 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 How to Deactivate It

Every command has a `\un-`version that deactivates it's functionality. So once you used `\chickenize`, it will chickenize the whole document up to `\unchickenize`. However, the paragraph in which `\unchickenize` appears, will *not* be chickenized. The same is true for all other manipulations. Take care that you don't `\un-`anything befor activating it, as this will result in an error.³

If you want to manipulate only a part of a paragraph, you have use the `\text-`version of the function, see below. However, feel free to set and unset every function at will at any place in your document.

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

³Which is so far not catchable due to missing functionality in `luatexbase`.

2.3 `\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 have⁴ 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.⁵

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.

2.4 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`.⁶ 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.

⁴If they don't have, I did miss that, sorry. Please inform me about such cases.

⁵On 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`!

⁶To be honest, this is just `\defd` to `\directlua`. One small advantage of this is that \TeX comments do work.

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"`.

leetable = **<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. `leetable[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`.

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 takes 200 letters for this change. 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 your rainbow will be.

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.

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.

colorexpan = **<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 T_EX file

```
1 \input{luatexbase.sty}
2 % read the Lua code first
3 \directlua{dofile("chickenize.lua")}
4 % then define the global macros. These affect the whole document and will stay active until the functions wi
5 \def\chickenize{
6   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize")}
7   luatexbase.add_to_callback("start_page_number",function() texio.write("[..status.total_pages) end ,"cst
8   luatexbase.add_to_callback("stop_page_number",function() texio.write(" chickens]") end,"cstoppage"))} %
9 \def\unchickenize{
```

```

10 \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","chickenize")
11 \directlua{luatexbase.remove_from_callback("start_page_number","cstarttpage")}
12 \directlua{luatexbase.remove_from_callback("stop_page_number","cstoppage")}}
13
14 \def\colorstretch{
15 \directlua{luatexbase.add_to_callback("post_linebreak_filter",colorstretch,"stretch_expansion")}}
16 \def\uncolorstretch{
17 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","colorstretch")}}
18
19 \def\leetspeak{
20 \directlua{luatexbase.add_to_callback("post_linebreak_filter",leet,"1337")}}
21 \def\unleetspeak{
22 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","1337")}}
23
24 \def\rainbowcolor{
25 \directlua{luatexbase.add_to_callback("post_linebreak_filter",rainbowcolor,"rainbowcolor")}
26 \directlua{rainbowcolor = true}}
27 \def\unrainbowcolor{
28 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","rainbowcolor")}
29 \directlua{rainbowcolor = false}}
30 \let\nyanize\rainbowcolor
31 \let\unnyanize\unrainbowcolor
32
33 \def\randomcolor{
34 \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"randomcolor")}}
35 \def\unrandomcolor{
36 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomcolor")}}
37
38 \def\randomfonts{
39 \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomfonts,"randomfonts")}}
40 \def\unrandomfonts{
41 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomfonts")}}
42
43 \def\randomuclc{
44 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",randomuclc,"randomuclc")}}
45 \def\unrandomuclc{
46 \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","randomuclc")}}
47
48 \def\uppercasecolor{
49 \directlua{luatexbase.add_to_callback("post_linebreak_filter",uppercasecolor,"uppercasecolor")}}
50 \def\unuppercasecolor{
51 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","uppercasecolor")}}

```

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

```

52 \newluaTeXattribute\leetattr
53 \newluaTeXattribute\randcolorattr
54 \newluaTeXattribute\randfontattr
55
56 \long\def\textleetspeak#1%
57 {\setluaTeXattribute\leetattr{42}#1\unsetluaTeXattribute\leetattr}

```

```

58 \long\def\textrandomcolor#1%
59   {\setluatexattribute\randcolorattr{42}#1\unsetluatexattribute\randcolorattr}
60 \long\def\textrandomfonts#1%
61   {\setluatexattribute\randfontsassr{42}#1\unsetluatexattribute\randfontsassr}
62 \long\def\textrandomfonts#1%
63   {\setluatexattribute\randfontsassr{42}#1\unsetluatexattribute\randfontsassr}

```

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.

```

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

```

5 L^AT_EX package

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

```

65 \input{chickenize}
66 \RequirePackage{
67   expl3,
68   xkeyval,
69   xparse
70 }

```

5.1 Definition of User-Level Macros

```

71 %% We want to "chickenize" figures, too. So ...
72 \DeclareDocumentCommand\includegraphics{0{ }m}{
73   \fbox{Chicken}   %% actually, I'd love to draw a mp graph showing a chicken ...
74 }
75 %% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
76
77 \ExplSyntaxOff   %% because of the : in the domain
78 \NewDocumentCommand\balmerpeak{G{}0{-4cm}}{
79   \begin{tikzpicture}
80     \hspace*{#2}   %% anyhow necessary to fix centering ... strange :(
81     \begin{axis}
82       [width=10cm,height=7cm,
83        xmin=-0.005,xmax=0.28,ymin=-0.05,ymax=1,
84        xtick={0,0.02,...,0.27},ytick=\empty,
85        /pgf/number format/precision=3,/pgf/number format/fixed,
86        tick label style={font=\small},
87        label style = {font=\Large},
88        xlabel = \fontspec{Punk Nova} BLOOD ALCOHOL CONCENTRATION (\%),
89        ylabel = \fontspec{Punk Nova} \rotatebox{-90}{\parbox{3cm}{\center programming\\ skills}}]
90       \addplot
91         [domain=-0.01:0.27,color=red,samples=250]
92         {0.8*exp(-0.5*((x-0.1335)^2)/.00002)+
93          0.5*exp(-0.5*((x+0.015)^2)/0.01)
94         };
95     \end{axis}

```

```

96 \end{tikzpicture}
97 }
98 \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.

```

99 Hhead = node.id("hhead")
100 RULE = node.id("rule")
101 GLUE = node.id("glue")
102 WHAT = node.id("whatsit")
103 COL = node.subtype("pdf_colorstack")
104 GLYPH = node.id("glyph")

```

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

```

105 color_push = node.new(WHAT,COL)
106 color_pop = node.new(WHAT,COL)
107 color_push.stack = 0
108 color_pop.stack = 0
109 color_push.cmd = 1
110 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.

```

111 chickenstring = "Chicken"
112
113 local tbl = font.getfont(font.current())
114 local space = tbl.parameters.space
115 local shrink = tbl.parameters.space_shrink
116 local stretch = tbl.parameters.space_stretch
117 local match = unicode.utf8.match
118
119 chickenize = function(head)
120   for i in node.traverse_id(37,head) do --find start of a word
121     while ((i.next.id == 37) or (i.next.id == 11) or (i.next.id == 7) or (i.next.id == 0)) do --find end of
122       i.next = i.next.next
123     end
124
125     chicken = {} -- constructing the node list. Should be done only once?
126     chicken[0] = node.new(37,1) -- only a dummy for the loop
127     for i = 1,string.len(chickenstring) do
128       chicken[i] = node.new(37,1)
129       chicken[i].font = font.current()

```



```

130     chicken[i-1].next = chicken[i]
131 end
132
133 j = 1
134 for s in string.utfvalues(chickenstring) do
135     local char = unicode.utf8.char(s)
136     chicken[j].char = s
137     if match(char,"%s") then
138         chicken[j] = node.new(10)
139         chicken[j].spec = node.new(47)
140         chicken[j].spec.width = space
141         chicken[j].spec.shrink = shrink
142         chicken[j].spec.stretch = stretch
143     end
144     j = j+1
145 end
146 node.slide(chicken[1])
147 lang.hyphenate(chicken[1])
148 chicken[1] = node.kerning(chicken[1])
149 chicken[1] = node.ligaturing(chicken[1])
150
151 node.insert_before(head,i,chicken[1])
152 chicken[1].next = chicken[2] -- seems to be necessary ... to be fixed
153 chicken[string.len(chickenstring)].next = i.next
154 end
155
156 return head
157 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.

```

158 leet_onlytext = false
159 leettable = {
160     [101] = 51, -- E
161     [105] = 49, -- I
162     [108] = 49, -- L
163     [111] = 48, -- O
164     [115] = 53, -- S
165     [116] = 55, -- T
166
167     [101-32] = 51, -- e
168     [105-32] = 49, -- i
169     [108-32] = 49, -- l
170     [111-32] = 48, -- o
171     [115-32] = 53, -- s
172     [116-32] = 55, -- t
173 }

```

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

```
174 leet = function(head)
175   for line in node.traverse_id(Hhead,head) do
176     for i in node.traverse_id(GLYPH,line.head) do
177       if not(leetspeak_onlytext) or
178         node.has_attribute(i,luatexbase.attributes.leetattr)
179       then
180         if leettable[i.char] then
181           i.char = leettable[i.char]
182         end
183       end
184     end
185   end
186   return head
187 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 explicitly in terms of `\bf` etc.

```
188 randomfontslower = 1
189 randomfontsupper = 0
190 %
191 randomfonts = function(head)
192   if (randomfontsupper > 0) then -- fixme: this should be done only once, no? Or at every paragraph?
193     rfub = randomfontsupper -- user-specified value
194   else
195     rfub = font.max() -- or just take all fonts
196   end
197   for line in node.traverse_id(Hhead,head) do
198     for i in node.traverse_id(GLYPH,line.head) do
199       if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattrib) then
200         i.font = math.random(randomfontslower,rfub)
201       end
202     end
203   end
204   return head
205 end
```

6.4 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

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

```

213 end
214 end
215 return head
216 end

```

6.5 randomchars

```

217 randomchars = function(head)
218   for line in node.traverse_id(Hhead,head) do
219     for i in node.traverse_id(GLYPH,line.head) do
220       i.char = math.floor(math.random()*512)
221     end
222   end
223   return head
224 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.

```

225 randomcolor_grey = false
226 randomcolor_onlytext = false --switch between local and global colorization
227 rainbowcolor = false
228
229 grey_lower = 0
230 grey_upper = 900
231
232 Rgb_lower = 1
233 rGb_lower = 1
234 rgB_lower = 1
235 Rgb_upper = 254
236 rGb_upper = 254
237 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.

```

238 rainbow_step = 0.005
239 rainbow_Rgb = 1-rainbow_step -- we start in the red phase
240 rainbow_rGb = rainbow_step -- values x must always be 0 < x < 1
241 rainbow_rgB = rainbow_step
242 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.

```

243 randomcolorstring = function()
244   if randomcolor_grey then
245     return (0.001*math.random(grey_lower,grey_upper)).." g"
246   elseif rainbowcolor then
247     if rainind == 1 then -- red
248       rainbow_rGb = rainbow_rGb + rainbow_step
249       if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
250     elseif rainind == 2 then -- yellow

```

```

251     rainbow_Rgb = rainbow_Rgb - rainbow_step
252     if rainbow_Rgb <= rainbow_step then rainind = 3 end
253 elseif rainind == 3 then -- green
254     rainbow_rGb = rainbow_rGb + rainbow_step
255     rainbow_rGb = rainbow_rGb - rainbow_step
256     if rainbow_rGb <= rainbow_step then rainind = 4 end
257 elseif rainind == 4 then -- blue
258     rainbow_Rgb = rainbow_Rgb + rainbow_step
259     if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
260 else -- purple
261     rainbow_rGb = rainbow_rGb - rainbow_step
262     if rainbow_rGb <= rainbow_step then rainind = 1 end
263 end
264 return rainbow_Rgb..rainbow_rGb..rainbow_rGb.." rg"
265 else
266     Rgb = math.random(Rgb_lower,Rgb_upper)/255
267     rGb = math.random(rGb_lower,rGb_upper)/255
268     rgB = math.random(rgB_lower,rgB_upper)/255
269     return Rgb..rGb..rgB.." rg"
270 end
271 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. Otherwise, all glyphs are taken.

```

272 randomcolor = function(head)
273   for line in node.traverse_id(0,head) do
274     for i in node.traverse_id(37,line.head) do
275       if not(randomcolor_onlytext) or
276         (node.has_attribute(i,luatexbase.attributes.randcolorattr))
277       then
278         color_push.data = randomcolorstring() -- color or grey string
279         line.head = node.insert_before(line.head,i,node.copy(color_push))
280         node.insert_after(line.head,i,node.copy(color_pop))
281       end
282     end
283   end
284   return head
285 end

```

6.7 uppercasecolor

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

```

286 uppercasecolor = function (head)
287   for line in node.traverse_id(Hhead,head) do
288     for upper in node.traverse_id(GLYPH,line.head) do
289       if (((upper.char > 64) and (upper.char < 91)) or
290         ((upper.char > 57424) and (upper.char < 57451))) then -- for small caps! nice
291         color_push.data = randomcolorstring() -- color or grey string

```

```

292     line.head = node.insert_before(line.head,upper,node.copy(color_push))
293     node.insert_after(line.head,upper,node.copy(color_pop))
294     end
295   end
296 end
297 return head
298 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 light gray, whereas a too dense line is indicated by a dark grey box.

The second box is only useful if microtypographic extensions are used, e. g. with the microtype package under L^AT_EX. 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.

```

299 keeptext = true
300 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.

```

301 colorstretch = function (head)
302
303   local f = font.getfont(font.current()).characters
304   for line in node.traverse_id(Hhead,head) do
305     local rule_bad = node.new(RULE)
306
307     if colorexpansion then -- if also the font expansion should be shown
308       local g = line.head
309       while not(g.id == 37) do
310         g = g.next
311       end
312       exp_factor = g.width / f[g.char].width
313       exp_color = .5 + (1-exp_factor)*10 .. " g"
314       rule_bad.width = 0.5*line.width -- we need two rules on each line!
315     else
316       rule_bad.width = line.width -- only the space expansion should be shown, only one rule
317     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.

```
318 rule_bad.height = tex.baselineskip.width*4/5 -- this should give a better output
319 rule_bad.depth = tex.baselineskip.width*1/5
320
321 local glue_ratio = 0
322 if line.glue_order == 0 then
323   if line.glue_sign == 1 then
324     glue_ratio = .5 * math.min(line.glue_set,1)
325   else
326     glue_ratio = -.5 * math.min(line.glue_set,1)
327   end
328 end
329 color_push.data = .5 + glue_ratio .. " g"
```

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

```
330 -- set up output
331 local p = line.head
332
333 -- a rule to immitate kerning all the way back
334 local kern_back = node.new(RULE)
335 kern_back.width = -line.width
336
337 -- if the text should still be displayed, the color and box nodes are inserted additionally
338 -- and the head is set to the color node
339 if keeptext then
340   line.head = node.insert_before(line.head,line.head,node.copy(color_push))
341 else
342   node.flush_list(p)
343   line.head = node.copy(color_push)
344 end
345 node.insert_after(line.head,line.head,rule_bad) -- then the rule
346 node.insert_after(line.head,line.head.next,node.copy(color_pop)) -- and then pop!
347 tmpnode = node.insert_after(line.head,line.head.next.next,kern_back)
348
349 -- then a rule with the expansion color
350 if colorexpan then -- if also the stretch/shrink of letters should be shown
351   color_push.data = exp_color
352   node.insert_after(line.head,tmpnode,node.copy(color_push))
353   node.insert_after(line.head,tmpnode.next,node.copy(rule_bad))
354   node.insert_after(line.head,tmpnode.next.next,node.copy(color_pop))
355 end
356 end
357 return head
358 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:

?