

*»The Monty Pythons, were they  $\TeX$  users,  
could have written the chickenize macro.«*

Paul Isambert

# CHICKENIZE

v0.1

Arno Trautmann

[arno.trautmann@gmx.de](mailto:arno.trautmann@gmx.de)

This is the documentation of the package `chickenize`. It allows manipulations of any Lua $\TeX$  document<sup>1</sup> exploiting the possibilities offered by the callbacks that influence line breaking. Most of this package's content is just for fun and educational use, but there are also some functions that can be useful in a normal document.

The table on the next page informs you shortly about some of your possibilities and provides links to the Lua functions. The  $\TeX$  interface is presented [below](#).

The documentation of this package is far from being well-readable, consistent or even complete. This is caused either by lack of time or priority. If you miss anything that should be documented or if you have suggestions on how to increase the readability of the descriptions, please let me know.

For a better understanding of what's going on in the code of this package, there is a small [tutorial](#) below that explains shortly the most important features used here.

*Attention:* This package is under development and everything presented here might be subject to incompatible changes. If, by any reason, you decide to use this package for an important document, please make a local copy of the source code and use that. This package will not be considered stable until it reaches at least v0.5.

If you have any suggestions or comments, just drop me a mail, I'll be happy to get any response! The latest source code is hosted on github: <https://github.com/alt/chickenize>. Feel free to comment or report bugs there, to fork, pull, etc.

This package is copyright © 20012 Arno L. Trautmann. It may be distributed and/or modified under the conditions of the LaTeX Project Public License, either version 1.3c of this license or (at your option) any later version. This work has the LPPL maintenance status ‘author-maintained’.

---

<sup>1</sup>The code is based on pure Lua $\TeX$  features, so don't even try to use it with any other  $\TeX$  flavour. The package is tested under plain Lua $\TeX$  and Lua $\LaTeX$ . If you tried using it with Con $\TeX$ t, please share your experience, I will gladly try to make it compatible!

---

**maybe useful functions**

---

<a href="#">colorstretch</a>	shows grey boxes that visualise the badness and font expansion of each line
<a href="#">letterspaceadjust</a>	uses a small amount of letterspacing to improve the greyness, especially for narrow lines

---

**less useful functions**

---

<a href="#">leetspeak</a>	translates the (latin-based) input into 1337 5p34k
<a href="#">randomucl</a>	alternates randomly between uppercase and lowercase
<a href="#">rainbowcolor</a>	changes the color of letters slowly according to a rainbow
<a href="#">randomcolor</a>	prints every letter in a random color
<a href="#">tabularasa</a>	removes every glyph from the output and leaves an empty document
<a href="#">uppercasecolor</a>	makes every uppercase letter colored

---

**complete nonsense**

---

<a href="#">chickenize</a>	replaces every word with “chicken”
<a href="#">gutenbergize</a>	deletes every quote and footnotes
<a href="#">hammertime</a>	U can't touch this!
<a href="#">kernmanipulation</a>	manipulates the kerning (tbi)
<a href="#">matrixize</a>	replaces every glyph by its ASCII value in binary code
<a href="#">randomerror</a>	just throws random (La)T <sub>E</sub> X errors at random times
<a href="#">randomfonts</a>	changes the font randomly between every letter
<a href="#">randomchars</a>	randomizes the (letters of the) whole input

---

# Contents

<b>I</b>	<b>User Documentation</b>	<b>5</b>
1	How It Works	5
2	Commands – How You Can Use It	5
2.1	TeX Commands – Document Wide	5
2.2	How to Deactivate It	6
2.3	\text-Versions	6
2.4	Lua functions	7
3	Options – How to Adjust It	7
<b>II</b>	<b>Tutorial</b>	<b>9</b>
4	Lua code	9
5	callbacks	9
5.1	How to use a callback	10
6	Nodes	10
7	Other things	11
<b>III</b>	<b>Implementation</b>	<b>12</b>
8	TeX file	12
9	TeX package	17
9.1	Definition of User-Level Macros	18
10	Lua Module	18
10.1	chickenize	19
10.2	guttenbergenize	21
10.2.1	guttenbergenize – preliminaries	21
10.2.2	guttenbergenize – the function	21
10.3	hammertime	22
10.4	itsame	22
10.5	kernmanipulate	23
10.6	leetspeak	24
10.7	letterspaceadjust	24
10.7.1	setup of variables	25
10.7.2	function implementation	25
10.8	matrixize	25

10.9	pancakenize	26
10.10	randomfonts	26
10.11	randomucl	26
10.12	randomchars	27
10.13	randomcolor and rainbowcolor	27
10.13.1	randomcolor – preliminaries	27
10.13.2	randomcolor – the function	28
10.14	randomerror	29
10.15	rickroll	29
10.16	tabularasa	29
10.17	uppercasecolor	29
10.18	colorstretch	30
10.18.1	colorstretch – preliminaries	30
10.19	zebranize	33
10.19.1	zebranize – preliminaries	33
10.19.2	zebranize – the function	33
<b>11</b>	<b>Drawing</b>	<b>35</b>
<b>12</b>	<b>Known Bugs</b>	<b>37</b>
<b>13</b>	<b>To Dos</b>	<b>37</b>
<b>14</b>	<b>Literature</b>	<b>37</b>
<b>15</b>	<b>Thanks</b>	<b>37</b>

## Part I

# User Documentation

## 1 How It Works

We make use of Lua $\TeX$ s callbacks, especially the `pre_linebreak_filter` and the `post_linebreak_filter`. Hooking a function into these, we can nearly arbitrarily change the content of the document. If the changes should be on the input-side (e. g. replacing words 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 to be preferred for such things.

All functions traverse the node list of a paragraph and manipulate the nodes' properties (like `.font` or `.char`) or insert nodes (like `color push/pop` nodes) and return this changed node list.

## 2 Commands – How You Can Use It

There are several ways to make use of the *chickenize* 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 simple 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 `\chickenize` setup described [below](#).

**`\chickenize`** Replaces every word of the input with the word “chicken”. Maybe sometime the replacement will be made configurable, but up to now, it's only chicken. To be a bit less static, about every 10<sup>th</sup> chicken is uppercase. However, the beginning of a sentence is not recognized automatically.<sup>2</sup>

**`\dubstepize`** wub wub wub wub wub BROOOOOAR WOBBBWOBWOB BZZZZRRRRRRROOOOOOAAAAA  
... (inspired by <http://www.youtube.com/watch?v=ZFQ5Ep07iHk> and <http://www.youtube.com/watch?v=nGxpSsbodnw>)

**`\dubstepenize`** synonym for `\dubstepize` as I am not sure what is the better name. Both macros are just a special case of `chickenize` with a very special “zoo” ... there is no `\undubstepize` – once you go `dubstep`, you cannot go back ...

**`\hammertime`** STOP! — Hammertime!

**`\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.

**`\randomerror`** Just throws a random  $\TeX$  or  $\LaTeX$  error at a random time during the compilation. I have quite no idea what this could be used for.

---

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

- `\randomucl` 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 its name says.
- `\rainbowcolor` Instead of random colors, this command causes the text color to change gradually according to the colors of a rainbow. Do not mix this with `randomcolor`, as that doesn't make any sense.
- `\pancakelize` This is a dummy command that does nothing. However, every time you use it, you owe a pancake to the package author. You can either send it via mail or bring it to some (local) T<sub>E</sub>X user's group meeting.
- `\tabularasa` Takes every glyph out of the document and replaces it by empty space of the same width. That could be useful if you want to hide some part of a text or similar. The `\text`-version is most likely more useful.
- `\leetspeak` Translates the input into 1337 speak. If you don't understand that, lern it, n00b.
- `\nyanize` A synonym for `rainbowcolor`.
- `\matrixize` Replaces every glyph by a binary representation of its ASCII value.
- `\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 greyness of both boxes indicate how well the greyness is distributed over the typeset page.

## 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 bevor activating it, as this will result in an error.<sup>3</sup>

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

## 2.3 \text-Versions

The functions provided by 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<sup>4</sup> a `\text`-version that takes an argument. `\textrandomcolor{foo}` results in a colored `foo` while the rest of the document remains unaffected. However, to achieve this effect, still the whole

---

<sup>3</sup>Which is so far not catchable due to missing functionality in luatexbase.

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

node list has to be traversed. Thus, it may slow down the compilation of your document, even if you use `\textrandomcolor` only once. Fortunately, the effect is very small and mostly negligible.<sup>5</sup>

Please don't fool around by mixing a `\text`-version with the non-`\text`-version. If you feel like it and are not pleased 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 independently. 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 (here: `chickenize`) specifies the function name; the available functions are listed below. You can supply a label as you like in the third argument. The fourth and last argument, which is omitted in the example, determines 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 Options – 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`. But be *careful*! The argument of `\chickenizesetup` is passed 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 *only* % as comment string. If you use `--`, all of the argument will be ignored as  $\TeX$  does not pass an eol to `\directlua`. If you don't understand that, just ignore it and go on as usual.

The following list tries to kind of keep track of the options and variables. There is no guarantee for completeness, 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 around with them a bit to find out what they are doing.

**chickenstring = <table>** The string that is printed when using `\chickenize`. In fact, `chickenstring` is a table which allows for some more random action. To specify the default string, say `chickenstring[1] = 'chicken'`. For more than one animal, just step the index: `chickenstring[2] = 'rabbit'`. All existing table entries will be used randomly. Remember that we are dealing with Lua strings here, so use `' '` to mark them. (`" "` can cause problems with `babel`.)

---

<sup>5</sup>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`!

`chickenizefraction = <float> 1` Gives the fraction of words that get replaced by the `chickenstring`. The default means that every word is substituted. However, with a value of, say, 0.0001, only one word in ten thousand will be `chickenstring`. `chickenizefraction` must be specified *after* `\begin{document}`. No idea, why ...

`chickencount = <true>` Activates the counting of substituted words and prints the number at the end of the terminal output.

`colorstretchnumbers = <true> 0` If true, the amount of stretching or shrinking of each line is printed into the margin as a green, red or black number.

`chickenkernamount = <int>` The amount the kerning is set to when using `\kernmanipulate`.

`chickenkerninvert = <bool>` If set to true, the kerning is inverted (to be used with `\kernmanipulate`).

`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`.

`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 the transition to be completed. 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 range, 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.

`colorexansion = <bool> true` If true, two bars are shown of which the second one denotes the font expansion. Only useful if font expansion is used. (You *do* use font expansion, don't you?)



## Part II

# Tutorial

I thought it might be helpful to add a small tutorial to this package as it is mainly written with instructional purposes in mind. However, the following is *not* intended as a comprehensive guide to Lua<sub>TeX</sub>! It's just to get an idea how things work here. For a deeper understanding of Lua<sub>TeX</sub> you should consult both the Lua<sub>TeX</sub> manual and some introduction into Lua proper like “Programming in Lua”. (See the section [Literature](#) at the end of the manual.)

## 4 Lua code

The crucial novelty in Lua<sub>TeX</sub> is the first part of its name: The programming language Lua. One can use nearly any Lua code inside the commands `\directlua{}` or `\latelua{}`. This alleviates simple tasks like calculating a number and printing it, just as if it was entered by hand:

```
\directlua{
  a = 5*2
  tex.print(a)
}
```

A number of additions to the Lua language renders it particularly suitable for <sub>TeX</sub>ing, especially the `tex.` library that offers access to <sub>TeX</sub> internals. In the simple example above, the function `tex.print()` inserts its argument into the <sub>TeX</sub> input stream, so the result of the calculation (10) is printed in the document.

Larger parts of Lua code should not be embedded in your <sub>TeX</sub> code, but rather in a separate file. It can then be loaded using

```
\directlua{dofile("filename")}
```

If you use Lua<sub>TeX</sub>, you can also use the `luacode` environment from the eponymous package.

## 5 callbacks

While Lua code can be inserted using `\directlua` at any point in the input, a very powerful concept allows to change the way <sub>TeX</sub> behaves: The *callbacks*. A callback is a point where you can hook into <sub>TeX</sub>'s working and do anything to it that may make sense – or not. (Thus maybe breaking your document completely ...)

Callbacks are employed at several stages of <sub>TeX</sub>'s work – e. g. for font loading, paragraph breaking, shipping out etc. In this package, we make heavy use of mostly two callbacks: The `pre_linebreak_filter` and the `post_linebreak` filter. These callbacks are called just before (or after, resp.) <sub>TeX</sub> breaks a paragraph into lines. Normally, these callbacks are empty, so they are a great playground. In between these callbacks, the `linebreak_filter` takes care of <sub>TeX</sub>'s line breaking mechanism. We won't touch this as I have no idea of what's going on there ;)

## 5.1 How to use a callback

The normal way to use a callback is to “register” a function in it. This way, the function is called each time the callback is executed. Typically, the function takes a node list (see below) as an argument, does something with it, and returns it. So a basic use of the `post_linebreak_filter` would look like:

```
function my_new_filter(head)
  return head
end
```

```
callback.register("post_linebreak_filter",my_new_filter)
```

The function `callback.register` takes the name of the callback and your new function. However, there are some reasons why we avoid this syntax here. Instead, we rely on the package `luatexbase` by Manuel Pégourié-Gonnard and Élie Roux that offers the function `luatexbase.add_to_callback` which has a somewhat extended syntax:

```
luatexbase.add_to_callback("post_linebreak_filter",my_new_filter,"a fancy new filter")
```

The third argument is a name you can (have to) give to your function in the callback. That is necessary because the package also allows for removing functions from callbacks, and then you need a unique identifier for the function:

```
luatexbase.remove_from_callback("post_linebreak_filter","a fancy new filter")
```

You have to consult the LuaTeX manual to see what functionality a callback has when executed, what arguments it expects and what return values have to be given.

Everything I have written here is not the complete truth – please consult the LuaTeX manual and the `luatexbase` documentation for details!

## 6 Nodes

Essentially everything that LuaTeX deals with are nodes – letters, spaces, colors, rules etc. In this package, we make heavy use of different types of nodes, so an understanding of the concept is crucial for the functionality.

A node is an object that has different properties, depending on its type which is stored in its `.id` field. For example, a node of type `glyph` has `id 37`, has a number `.char` that represents its unicode codepoint, a `.font` entry that determines the font used for this glyph, a `.height`, `.depth` and `.width` etc.

Also, a node typically has a non-empty field `.next` and `.prev`. In a list, these point to the – guess it – next or previous node. Using this, one can walk over a list of nodes step by step and manipulate the list.

A more convenient way to adress each node of a list is the function `node.traverse(head)` which takes as first argument the first node of the list. However, often one wants to adress only a certain type of nodes in a list – e. g. all glyphs in a vertical list that also contains glue, rules etc. This is achieved by calling the function `node.traverse_id(37,head)`, with the first argument giving the respective id of the nodes.

The following example removes all characters “e” from the input just before paragraph breaking. This might not make any sense, but it is a good example anyways:

```
function remove_e(head)
```

```

for n in node.traverse_id(37,head) do
  if n.char == 101 then
    node.remove(head,n)
  end
end
return head
end

luatexbase.add_to_callback("pre_linebreak_filter",remove_e,"remove all letters e")

```

Now, don't read on, but try out this code by yourself! Change the number of the character to be removed, try to play around a bit. Also, try to remove the spaces between words. Those are glue nodes – look up their id in the Lua<sub>T</sub><sub>E</sub>X manual! Then, you have to remove the `if n.char` condition on the third line of the listing, because glue nodes lack a `.char` field. If everything works, you should have an input consisting of only one long word. Congratulations!

The `pre_linebreak_filter` is especially easy because its argument (here called `head`) is just one horizontal list. For the `post_linebreak_filter`, one has to traverse a whole vertical stack of horizontal lists, vertical glue and other material. See some of the functions below to understand what is necessary in this more complicated case.

## 7 Other things

Lua is a very intuitive and simple language, but nonetheless powerful. Just two tips: use local variables if possible – your code will be much faster. For this reason we prefer synonyms like `nodetraverseid = node.traverse_id` instead of the original names.

Also, Lua is kind of built around tables. Everything is best done with tables!

The namespace of the `chickenize` package is *not* consistent. Please don't take anything here as an example for good Lua coding, for good  $\TeX$ ing or even for good Lua<sub>T</sub><sub>E</sub>Xing. It's not. For high quality code check out the code written by Hans Hagen or other professionals. Once you understand the package at hand, you should be ready to go on and improve your knowledge. After that, you might come back and help me improve this package – I'm always happy for any help ☺

## Part III

# Implementation

## 8 $\text{\TeX}$ file

This file is more-or-less a dummy file to offer a nice interface for the functions. Basically, every macro registers a function of the same name in the corresponding callback. The un-macros later remove these functions. Where it makes sense, there are text-variants that activate the function only in a certain area of the text, by means of Lua $\text{\TeX}$ 's attributes.

For (un)registering, we use the `luatexbase` package. Then, the `.lua` file is loaded which does the actual work. Finally, the  $\text{\TeX}$  macros are defined as simple `\directlua` calls.

The Lua file is not found by using a simple `dofile("chickenize.lua")` call, but we have to use `kpse's find_file`.

```
1 \input{luatexbase.sty}
2 \directlua{dofile(kpse.find_file("chickenize.lua"))}
3
4 \def\chickenize{
5   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize")}
6   luatexbase.add_to_callback("start_page_number",
7     function() texio.write("[..status.total_pages) end ","cstartpage")
8   luatexbase.add_to_callback("stop_page_number",
9     function() texio.write(" chickens]") end,"cstoppage")}
10 %
11   luatexbase.add_to_callback("stop_run",nicetext,"a nice text")
12 }
13 }
14 \def\unchickenize{
15   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","chickenize")}
16   luatexbase.remove_from_callback("start_page_number","cstartpage")
17   luatexbase.remove_from_callback("stop_page_number","cstoppage")}}
18
19 \def\coffeestainize{ %% to be implemented.
20   \directlua{}}
21 \def\uncoffeestainize{
22   \directlua{}}
23
24 \def\colorstretch{
25   \directlua{luatexbase.add_to_callback("post_linebreak_filter",colorstretch,"stretch_expansion")}
26 \def\uncolorstretch{
27   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","stretch_expansion")}}
28
29 \def\dosomethingfunny{
30   %% should execute one of the "funny" commands, but randomly. So every compilation is complete.
31 }
```

```

32
33 \def\dubstepenize{
34   \chickenize
35   \directlua{
36     chickenstring[1] = "WOB"
37     chickenstring[2] = "WOB"
38     chickenstring[3] = "WOB"
39     chickenstring[4] = "BROOOAR"
40     chickenstring[5] = "WHEE"
41     chickenstring[6] = "WOB WOB WOB"
42     chickenstring[7] = "WAAAAAAAAAH"
43     chickenstring[8] = "duhduh duhduh duh"
44     chickenstring[9] = "BEEEEEEEEEW"
45     chickenstring[10] = "DEEEEEEEEEW"
46     chickenstring[11] = "EEEEEW"
47     chickenstring[12] = "boop"
48     chickenstring[13] = "buhdee"
49     chickenstring[14] = "bee bee"
50     chickenstring[15] = "BZZZRRRRRRR000000AAAAA"
51
52     chickenizefraction = 1
53   }
54 }
55 \let\dubstepize\dubstepenize
56
57 \def\gutenbergenize{ %% makes only sense when using LaTeX
58   \AtBeginDocument{
59     \let\grqq\relax\let\glqq\relax
60     \let\frqq\relax\let\flqq\relax
61     \let\grq\relax\let\glq\relax
62     \let\frq\relax\let\flq\relax
63   %%
64     \gdef\footnote##1{}
65     \gdef\cite##1{}\gdef\parencite##1{}
66     \gdef\Cite##1{}\gdef\Parencite##1{}
67     \gdef\cites##1{}\gdef\parencites##1{}
68     \gdef\Cites##1{}\gdef\Parencites##1{}
69     \gdef\footcite##1{}\gdef\footcitetext##1{}
70     \gdef\footcites##1{}\gdef\footcitetexts##1{}
71     \gdef\textcite##1{}\gdef\Textcite##1{}
72     \gdef\textcites##1{}\gdef\Textcites##1{}
73     \gdef\smartcites##1{}\gdef\Smartcites##1{}
74     \gdef\supercite##1{}\gdef\supercites##1{}
75     \gdef\autocite##1{}\gdef\Autocite##1{}
76     \gdef\autocites##1{}\gdef\Autocites##1{}
77     %% many, many missing ... maybe we need to tackle the underlying mechanism?

```

```

78 }
79 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",guttenbergenize_rq,"guttenbergenize_rq")}
80 }
81
82 \def\hammertime{
83   \global\let\n\relax
84   \directlua{hammerfirst = true
85             luatexbase.add_to_callback("pre_linebreak_filter",hammertime,"hammertime")}}
86 \def\unhammertime{
87   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","hammertime")}}
88
89 % \def\itsame{
90 %   \directlua{drawmario}} %% does not exist
91
92 \def\kernmanipulate{
93   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",kernmanipulate,"kernmanipulate")}}
94 \def\unkernmanipulate{
95   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter",kernmanipulate)}}
96
97 \def\leetspeak{
98   \directlua{luatexbase.add_to_callback("post_linebreak_filter",leet,"1337")}}
99 \def\unleetspeak{
100   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","1337")}}
101
102 \def\letterspaceadjust{
103   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",letterspaceadjust,"letterspaceadjust")}}
104 \def\unletterspaceadjust{
105   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter",letterspaceadjust)}}
106
107 \let\stealsheep\letterspaceadjust %% synonym in honor of Paul
108 \let\unstealsheep\unletterspaceadjust
109 \let\returnsheep\unletterspaceadjust
110
111 \def\matrixize{
112   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",matrixize,"matrixize")}}
113 \def\unmatrixize{
114   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter",matrixize)}}
115
116 \def\milkcow{ %% FIXME %% to be implemented
117   \directlua{}}
118 \def\unmilkcow{
119   \directlua{}}
120
121 \def\pancakenize{ %% FIXME %% to be implemented
122   \directlua{luatexbase.add_to_callback("stop_run",pancaketext,"pancaketext")}}
123 \def\unpancakenize{

```

```

124 \directlua{}}
125
126 \def\rainbowcolor{
127 \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"rainbowcolor")
128         rainbowcolor = true}}
129 \def\unrainbowcolor{
130 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","rainbowcolor")
131         rainbowcolor = false}}
132 \let\nyanize\rainbowcolor
133 \let\unnyanize\unrainbowcolor
134
135 \def\randomcolor{
136 \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"randomcolor")}}
137 \def\unrandomcolor{
138 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomcolor")}}
139
140 \def\randomerror{ %% FIXME
141 \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"randomcolor")}}
142 \def\unrandomerror{ %% FIXME
143 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomcolor")}}
144
145 \def\randomfonts{
146 \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomfonts,"randomfonts")}}
147 \def\unrandomfonts{
148 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomfonts")}}
149
150 \def\randomuclc{
151 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",randomuclc,"randomuclc")}}
152 \def\unrandomuclc{
153 \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","randomuclc")}}
154
155 \def\scorpionize{
156 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",scorpionize_color,"scorpionize_color")}}
157 \def\unscorpionize{
158 \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","scorpionize_color")}}
159
160 \def\spankmonkey{ %% to be implemented
161 \directlua{}}
162 \def\unspankmonkey{
163 \directlua{}}
164
165 \def\tabularasa{
166 \directlua{luatexbase.add_to_callback("post_linebreak_filter",tabularasa,"tabularasa")}}
167 \def\untabularasa{
168 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","tabularasa")}}
169

```

```

170 \def\uppercasecolor{
171   \directlua{luatexbase.add_to_callback("post_linebreak_filter",uppercasecolor,"uppercasecolor")}}
172 \def\unuppercasecolor{
173   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","uppercasecolor")}}
174
175 \def\zebranize{
176   \directlua{luatexbase.add_to_callback("post_linebreak_filter",zebranize,"zebranize")}}
177 \def\unzebranize{
178   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","zebranize")}}

```

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.

```

179 \newluatexattribute\leetattr
180 \newluatexattribute\randcolorattr
181 \newluatexattribute\randfontsaattr
182 \newluatexattribute\randucclattr
183 \newluatexattribute\tabularasaattr
184 \newluatexattribute\uppercasecolorattr
185
186 \long\def\textleetspeak#1%
187   {\setluatexattribute\leetattr{42}#1\unsetluatexattribute\leetattr}
188 \long\def\textrandomcolor#1%
189   {\setluatexattribute\randcolorattr{42}#1\unsetluatexattribute\randcolorattr}
190 \long\def\textrandomfontsa#1%
191   {\setluatexattribute\randfontsaattr{42}#1\unsetluatexattribute\randfontsaattr}
192 \long\def\textrandomfontsa#1%
193   {\setluatexattribute\randfontsaattr{42}#1\unsetluatexattribute\randfontsaattr}
194 \long\def\textrandomuccl#1%
195   {\setluatexattribute\randucclattr{42}#1\unsetluatexattribute\randucclattr}
196 \long\def\texttabularasa#1%
197   {\setluatexattribute\tabularasaattr{42}#1\unsetluatexattribute\tabularasaattr}
198 \long\def\textuppercasecolor#1%
199   {\setluatexattribute\uppercasecolorattr{42}#1\unsetluatexattribute\uppercasecolorattr}

```

Finally, a macro to control the setup. So far, it's only a wrapper that allows TeX-style comments to make the user feel more at home.

```

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

```

The following is the very first try of implementing a small drawing language in Lua. It draws a beautiful chicken.

```

201 \long\def\luadraw#1#2{%
202   \vbox to #1bp{%
203     \vfil
204     \luatexlatalua{pdf_print("q") #2 pdf_print("Q")}%
205   }%
206 }
207 \long\def\drawchicken{
208 \luadraw{90}{

```



```

209 kopf = {200,50} % Kopfmitte
210 kopf_rad = 20
211
212 d = {215,35} % Halsansatz
213 e = {230,10} %
214
215 korper = {260,-10}
216 korper_rad = 40
217
218 bein11 = {260,-50}
219 bein12 = {250,-70}
220 bein13 = {235,-70}
221
222 bein21 = {270,-50}
223 bein22 = {260,-75}
224 bein23 = {245,-75}
225
226 schnabel_oben = {185,55}
227 schnabel_vorne = {165,45}
228 schnabel_unten = {185,35}
229
230 flugel_vorne = {260,-10}
231 flugel_unten = {280,-40}
232 flugel_hinten = {275,-15}
233
234 sloppycircle(kopf,kopf_rad)
235 sloppyline(d,e)
236 sloppycircle(korper,korper_rad)
237 sloppyline(bein11,bein12) sloppyline(bein12,bein13)
238 sloppyline(bein21,bein22) sloppyline(bein22,bein23)
239 sloppyline(schnabel_vorne,schnabel_oben) sloppyline(schnabel_vorne,schnabel_unten)
240 sloppyline(flugel_vorne,flugel_unten) sloppyline(flugel_hinten,flugel_unten)
241 }
242 }

```

## 9 L<sup>A</sup>T<sub>E</sub>X package

I have decided to keep the L<sup>A</sup>T<sub>E</sub>X-part of this package as small as possible. So far, it does ... nothing useful, but it provides a `chickenize.sty` that loads `chickenize.tex` so the user can still say `\usepackage{chickenize}`. This file will never support package options!

Some code might be implemented to manipulate figures for full chickenization. However, I will *not* load any packages at this place, as loading of `expl3` or `TikZ` or whatever takes too much time for such a tiny package like this one. If you require any of the features presented here, you have to load the packages on your own. Maybe this will change.

```

243 \ProvidesPackage{chickenize}%

```

```

244 [2012/05/20 v0.1 chickenize package]
245 \input{chickenize}

```

## 9.1 Definition of User-Level Macros

```

246 %% We want to "chickenize" figures, too. So ...
247 \iffalse
248 \DeclareDocumentCommand\includegraphics{0}{m}{
249   \fbox{Chicken} %% actually, I'd love to draw an MP graph showing a chicken ...
250 }
251 %%%% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
252 %% So far, you have to load pgfplots yourself.
253 %% As it is a mighty package, I don't want the user to force loading it.
254 \NewDocumentCommand\balmerpeak{G{}0{-4cm}}{
255 %% to be done using Lua drawing.
256 }
257 \fi

```

## 10 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 at the document level, too.

```

258
259 local nodenew = node.new
260 local nodecopy = node.copy
261 local nodeinsertbefore = node.insert_before
262 local nodeinsertafter = node.insert_after
263 local noderemove = node.remove
264 local nodeid = node.id
265 local nodetraverseid = node.traverse_id
266
267 Hhead = nodeid("hhead")
268 RULE = nodeid("rule")
269 GLUE = nodeid("glue")
270 WHAT = nodeid("whatsit")
271 COL = node.subtype("pdf_colorstack")
272 GLYPH = nodeid("glyph")

```

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

```

273 color_push = nodenew(WHAT,COL)
274 color_pop = nodenew(WHAT,COL)
275 color_push.stack = 0
276 color_pop.stack = 0
277 color_push.cmd = 1
278 color_pop.cmd = 2

```

## 10.1 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.

```
279 chicken_pagenumbers = true
280
281 chickenstring = {}
282 chickenstring[1] = "Chicken" -- chickenstring is a table, please remeber this!
283
284 chickenizefraction = 0.5
285 -- set this to a small value to fool somebody, or to see if your text has been read carefully. The
286 chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing your
287
288 local tbl = font.getfont(font.current())
289 local space = tbl.parameters.space
290 local shrink = tbl.parameters.space_shrink
291 local stretch = tbl.parameters.space_stretch
292 local match = unicode.utf8.match
293 chickenize_ignore_word = false
294
295 chickenize_real_stuff = function(i,head)
296     while ((i.next.id == 37) or (i.next.id == 11) or (i.next.id == 7) or (i.next.id == 0)) do --
297         i.next = i.next.next
298     end
299
300     chicken = {} -- constructing the node list.
301
302 -- Should this be done only once? No, otherwise we lose the freedom to change the string in-docum
303 -- But it could be done only once each paragraph as in-paragraph changes are not possible!
304
305     chickenstring_tmp = chickenstring[math.random(1,#chickenstring)]
306     chicken[0] = nodenew(37,1) -- only a dummy for the loop
307     for i = 1,string.len(chickenstring_tmp) do
308         chicken[i] = nodenew(37,1)
309         chicken[i].font = font.current()
310         chicken[i-1].next = chicken[i]
311     end
312
313     j = 1
314     for s in string.utfvalues(chickenstring_tmp) do
315         local char = unicode.utf8.char(s)
316         chicken[j].char = s
317         if match(char,"%s") then
318             chicken[j] = nodenew(10)
319             chicken[j].spec = nodenew(47)
```

```

320         chicken[j].spec.width = space
321         chicken[j].spec.shrink = shrink
322         chicken[j].spec.stretch = stretch
323     end
324     j = j+1
325 end
326
327 node.slide(chicken[1])
328 lang.hyphenate(chicken[1])
329 chicken[1] = node.kerning(chicken[1])    -- FIXME: does not work
330 chicken[1] = node.ligaturing(chicken[1]) -- dito
331
332 nodeinsertbefore(head,i,chicken[1])
333 chicken[1].next = chicken[2] -- seems to be necessary ... to be fixed
334 chicken[string.len(chickenstring_tmp)].next = i.next
335 return head
336 end
337
338 chickenize = function(head)
339   for i in nodetraverseid(37,head) do --find start of a word
340     if (chickenize_ignore_word == false) then -- normal case: at the beginning of a word, we jump
341       head = chickenize_real_stuff(i,head)
342     end
343
344 -- At the end of the word, the ignoring is reset. New chance for everyone.
345     if not((i.next.id == 37) or (i.next.id == 7) or (i.next.id == 22) or (i.next.id == 11)) then
346       chickenize_ignore_word = false
347     end
348
349 -- And the random determination of the chickenization of the next word:
350     if math.random() > chickenizefraction then
351       chickenize_ignore_word = true
352     elseif chickencount then
353       chicken_substitutions = chicken_substitutions + 1
354     end
355   end
356   return head
357 end
358
359 local separator      = string.rep("=", 28)
360 local texiowrite_nl = texio.write_nl
361 nicetext = function()
362   texiowrite_nl("Output written on "..tex.jobname.."pdf ("..status.total_pages.." chicken,".." eg
363   texiowrite_nl(" ")
364   texiowrite_nl(separator)
365   texiowrite_nl("Hello my dear user,")

```

```

366 texiowrite_nl("good job, now go outside and enjoy the world!")
367 texiowrite_nl(" ")
368 texiowrite_nl("And don't forget to feed your chicken!")
369 texiowrite_nl(separator .. "\n")
370 if chickencount then
371     texiowrite_nl("There were "..chicken_substitutions.." substitutions made.")
372     texiowrite_nl(separator)
373 end
374 end

```

## 10.2 guttenbergenize

A function in honor of the German politician Guttenberg.<sup>6</sup> Please do *not* confuse him with the grand master Gutenberg!

Calling `\guttenbergenize` will not only execute or manipulate Lua code, but also redefine some  $\TeX$  or  $\LaTeX$  commands. The aim is to remove all quotations, footnotes and anything that will give information about the real sources of your work.

The following Lua function will remove all quotation marks from the input. Again, the `pre_linebreak_filter` is used for this, although it should be rather removed in the input filter or so.

### 10.2.1 guttenbergenize – preliminaries

This is a nice solution Lua offers for our needs. Learn it, this might be helpful for you sometime, too.

```

375 local quotestrings = {
376     [171] = true, [172] = true,
377     [8216] = true, [8217] = true, [8218] = true,
378     [8219] = true, [8220] = true, [8221] = true,
379     [8222] = true, [8223] = true,
380     [8248] = true, [8249] = true, [8250] = true,
381 }

```

### 10.2.2 guttenbergenize – the function

```

382 guttenbergenize_rq = function(head)
383     for n in nodetraverseid(nodeid"glyph",head) do
384         local i = n.char
385         if quotestrings[i] then
386             noderemove(head,n)
387         end
388     end
389     return head
390 end

```

---

<sup>6</sup>Thanks to Jasper for bringing me to this idea!

### 10.3 hammertime

This is a completely useless function. It just prints STOP! – HAMMERTIME at the beginning of the first paragraph after `\hammertime`, and “U can't touch this” for every following one. As the function writes to the terminal, you have to be sure that your terminal is line-buffered and not block-buffered. Compare the explanation by Taco on the LuaTeX mailing list.<sup>7</sup>

```
391 hammertimedelay = 1.2
392 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
393 hammertime = function(head)
394   if hammerfirst then
395     texiowrite_nl(htime_separator)
396     texiowrite_nl("=====STOP!=====\\n")
397     texiowrite_nl(htime_separator .. "\\n\\n\\n")
398     os.sleep (hammertimedelay*1.5)
399     texiowrite_nl(htime_separator .. "\\n")
400     texiowrite_nl("=====HAMMERTIME=====\\n")
401     texiowrite_nl(htime_separator .. "\\n\\n")
402     os.sleep (hammertimedelay)
403     hammerfirst = false
404   else
405     os.sleep (hammertimedelay)
406     texiowrite_nl(htime_separator)
407     texiowrite_nl("=====U can't touch this!=====\\n")
408     texiowrite_nl(htime_separator .. "\\n\\n")
409     os.sleep (hammertimedelay*0.5)
410   end
411   return head
412 end
```

### 10.4 itsame

The (very first, very basic, very stupid) code to draw a small mario. You need to input `luadraw.tex` or do `luadraw.lua` for the rectangle function.

```
413 itsame = function()
414 local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
415 color = "1 .6 0"
416 for i = 6,9 do mr(i,3) end
417 for i = 3,11 do mr(i,4) end
418 for i = 3,12 do mr(i,5) end
419 for i = 4,8 do mr(i,6) end
420 for i = 4,10 do mr(i,7) end
421 for i = 1,12 do mr(i,11) end
422 for i = 1,12 do mr(i,12) end
423 for i = 1,12 do mr(i,13) end
424
```

---

<sup>7</sup><http://tug.org/pipermail/luatex/2011-November/003355.html>

```

425 color = ".3 .5 .2"
426 for i = 3,5 do mr(i,3) end mr(8,3)
427 mr(2,4) mr(4,4) mr(8,4)
428 mr(2,5) mr(4,5) mr(5,5) mr(9,5)
429 mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
430 for i = 3,8 do mr(i,8) end
431 for i = 2,11 do mr(i,9) end
432 for i = 1,12 do mr(i,10) end
433 mr(3,11) mr(10,11)
434 for i = 2,4 do mr(i,15) end for i = 9,11 do mr(i,15) end
435 for i = 1,4 do mr(i,16) end for i = 9,12 do mr(i,16) end
436
437 color = "1 0 0"
438 for i = 4,9 do mr(i,1) end
439 for i = 3,12 do mr(i,2) end
440 for i = 8,10 do mr(5,i) end
441 for i = 5,8 do mr(i,10) end
442 mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
443 for i = 4,9 do mr(i,12) end
444 for i = 3,10 do mr(i,13) end
445 for i = 3,5 do mr(i,14) end
446 for i = 7,10 do mr(i,14) end
447 end

```

## 10.5 kernmanipulate

This function either eliminates all the kerning, inverts the sign of the kerning or changes it to a user-given value.

If the boolean `chickeninvertkerning` is true, the kerning amount is negative, if it is false, the kerning will be set to the value of `chickenkernvalue`. A large value (> 100 000) can be used to show explicitly where kerns are inserted. Good for educational use.

```

448 chickenkernamount = 0
449 chickeninvertkerning = false
450
451 function kernmanipulate (head)
452   if chickeninvertkerning then -- invert the kerning
453     for n in nodetraverseid(11,head) do
454       n.kern = -n.kern
455     end
456   else -- if not, set it to the given value
457     for n in nodetraverseid(11,head) do
458       n.kern = chickenkernamount
459     end
460   end
461   return head
462 end

```

## 10.6 leetspeak

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

```
463 leetspeak_onlytext = false
464 leetable = {
465   [101] = 51, -- E
466   [105] = 49, -- I
467   [108] = 49, -- L
468   [111] = 48, -- O
469   [115] = 53, -- S
470   [116] = 55, -- T
471
472   [101-32] = 51, -- e
473   [105-32] = 49, -- i
474   [108-32] = 49, -- l
475   [111-32] = 48, -- o
476   [115-32] = 53, -- s
477   [116-32] = 55, -- t
478 }
```

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

```
479 leet = function(head)
480   for line in nodetraverseid(Hhead,head) do
481     for i in nodetraverseid(GLYPH,line.head) do
482       if not leetspeak_onlytext or
483         node.has_attribute(i,luatexbase.attributes.leetattr)
484       then
485         if leetable[i.char] then
486           i.char = leetable[i.char]
487         end
488       end
489     end
490   end
491   return head
492 end
```

## 10.7 letterspaceadjust

Yet another piece of code by Paul. This is primarily intended for very narrow columns, but may also increase the overall quality of typesetting. Basically, it does nothing else than adding expandable space *between* letters. This way, the amount of stretching between words can be reduced which will, hopefully, result in the greyness to be more equally distributed over the page.

Why the synonym `stealsheep`? Because of a comment of Paul on the `texhax` mailing list: <http://tug.org/pipermail/texhax/2011-October/018374.html>



### 10.7.1 setup of variables

```
493 local letterspace_glue = nodenew(nodeid"glue")
494 local letterspace_spec = nodenew(nodeid"glue_spec")
495 local letterspace_pen = nodenew(nodeid"penalty")
496
497 letterspace_spec.width = tex.sp"0pt"
498 letterspace_spec.stretch = tex.sp"2pt"
499 letterspace_glue.spec = letterspace_spec
500 letterspace_pen.penalty = 10000
```

### 10.7.2 function implementation

```
501 letterspaceadjust = function(head)
502   for glyph in nodetraverseid(nodeid"glyph", head) do
503     if glyph.prev and (glyph.prev.id == nodeid"glyph" or glyph.prev.id == nodeid"disc") then
504       local g = nodecopy(letterspace_glue)
505       nodeinsertbefore(head, glyph, g)
506       nodeinsertbefore(head, g, nodecopy(letterspace_pen))
507     end
508   end
509   return head
510 end
```

## 10.8 matrixize

Substitutes every glyph by a representation of its ASCII value. Might be extended to cover the entire unicode range, but so far only 8bit is supported. The code is quite straight-forward and works OK. The line ends are not necessarily adjusted correctly. However, with microtype, i. e. font expansion, everything looks fine.

```
511 matrixize = function(head)
512   x = {}
513   s = nodenew(nodeid"disc")
514   for n in nodetraverseid(nodeid"glyph",head) do
515     j = n.char
516     for m = 0,7 do -- stay ASCII for now
517       x[7-m] = nodecopy(n) -- to get the same font etc.
518     end
519     if (j / (2^(7-m)) < 1) then
520       x[7-m].char = 48
521     else
522       x[7-m].char = 49
523       j = j-(2^(7-m))
524     end
525     nodeinsertbefore(head,n,x[7-m])
526     nodeinsertafter(head,x[7-m],nodecopy(s))
527   end
528   noderemove(head,n)
529 end
```

```

530 return head
531 end

```

## 10.9 pancakenize

```

532 local separator      = string.rep("=", 28)
533 local texiowrite_nl = texio.write_nl
534 pancaketext = function()
535   texiowrite_nl("Output written on "..tex.jobname.." pdf ("..status.total_pages.." chicken,".." eg
536   texiowrite_nl(" ")
537   texiowrite_nl(separator)
538   texiowrite_nl("Soo ... you decided to use \\pancakenize.")
539   texiowrite_nl("That means you owe me a pancake!")
540   texiowrite_nl(" ")
541   texiowrite_nl("(This goes by document, not compilation.)")
542   texiowrite_nl(separator.."\\n\\n")
543   texiowrite_nl("Looking forward for my pancake! :)")
544 end

```

## 10.10 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.

```

545 local randomfontslower = 1
546 local randomfontsupper = 0
547 %
548 randomfonts = function(head)
549   local rfub
550   if randomfontsupper > 0 then -- fixme: this should be done only once, no? Or at every paragraph
551     rfub = randomfontsupper -- user-specified value
552   else
553     rfub = font.max() -- or just take all fonts
554   end
555   for line in nodetraverseid(Hhead,head) do
556     for i in nodetraverseid(GLYPH,line.head) do
557       if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattrib) then
558         i.font = math.random(randomfontslower,rfub)
559       end
560     end
561   end
562   return head
563 end

```

## 10.11 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```

564 uclcratio = 0.5 -- ratio between uppercase and lower case

```

```

565 randomuclc = function(head)
566   for i in nodetraverseid(37,head) do
567     if not(randomuclc_onlytext) or node.has_attribute(i,luatexbase.attributes.randuclcatr) then
568       if math.random() < uclcratio then
569         i.char = tex.uccode[i.char]
570       else
571         i.char = tex.lccode[i.char]
572       end
573     end
574   end
575   return head
576 end

```

## 10.12 randomchars

```

577 randomchars = function(head)
578   for line in nodetraverseid(Hhead,head) do
579     for i in nodetraverseid(GLYPH,line.head) do
580       i.char = math.floor(math.random()*512)
581     end
582   end
583   return head
584 end

```

## 10.13 randomcolor and rainbowcolor

### 10.13.1 randomcolor – preliminaries

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.

```

585 randomcolor_grey = false
586 randomcolor_onlytext = false --switch between local and global colorization
587 rainbowcolor = false
588
589 grey_lower = 0
590 grey_upper = 900
591
592 Rgb_lower = 1
593 rGb_lower = 1
594 rgB_lower = 1
595 Rgb_upper = 254
596 rGb_upper = 254
597 rgB_upper = 254

```

Variables for the rainbow.  $1/\text{rainbow\_step} \times 5$  is the number of letters used for one cycle, the color changes from red to yellow to green to blue to purple.

```

598 rainbow_step = 0.005
599 rainbow_Rgb = 1-rainbow_step -- we start in the red phase

```

```

600 rainbow_rGb = rainbow_step -- values x must always be 0 < x < 1
601 rainbow_rgB = rainbow_step
602 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.

```

603 randomcolorstring = function()
604   if randomcolor_grey then
605     return (0.001*math.random(grey_lower, grey_upper)).." g"
606   elseif rainbowcolor then
607     if rainind == 1 then -- red
608       rainbow_rGb = rainbow_rGb + rainbow_step
609       if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
610     elseif rainind == 2 then -- yellow
611       rainbow_Rgb = rainbow_Rgb - rainbow_step
612       if rainbow_Rgb <= rainbow_step then rainind = 3 end
613     elseif rainind == 3 then -- green
614       rainbow_rgB = rainbow_rgB + rainbow_step
615       rainbow_rGb = rainbow_rGb - rainbow_step
616       if rainbow_rGb <= rainbow_step then rainind = 4 end
617     elseif rainind == 4 then -- blue
618       rainbow_Rgb = rainbow_Rgb + rainbow_step
619       if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
620     else -- purple
621       rainbow_rgB = rainbow_rgB - rainbow_step
622       if rainbow_rgB <= rainbow_step then rainind = 1 end
623     end
624     return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rgB.." rg"
625   else
626     Rgb = math.random(Rgb_lower, Rgb_upper)/255
627     rGb = math.random(rGb_lower, rGb_upper)/255
628     rgB = math.random(rgB_lower, rgB_upper)/255
629     return Rgb.." "..rGb.." "..rgB.." " rg
630   end
631 end

```

### 10.13.2 randomcolor – the function

The function that does all the coloring 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.

```

632 randomcolor = function(head)
633   for line in nodetraverseid(0, head) do
634     for i in nodetraverseid(37, line.head) do
635       if not(randomcolor_onlytext) or
636         (node.has_attribute(i, luatexbase.attributes.randcolorattr))
637       then
638         color_push.data = randomcolorstring() -- color or grey string

```

```

639         line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
640         nodeinsertafter(line.head,i,nodecopy(color_pop))
641     end
642 end
643 end
644 return head
645 end

```

#### 10.14 randomerror

```

646 %

```

#### 10.15 rickroll

Another tribute to pop culture. Either: substitute word-by-word as in pancake. OR: substitute each link to a youtube-rickroll ...

#### 10.16 tabularasa

Removes every glyph from the output and replaces it by empty space. In the end, next to nothing will be visible. Should be extended to also remove rules or just anything visible.

```

647 tabularasa_onlytext = false
648
649 tabularasa = function(head)
650     local s = nodenew(nodeid"kern")
651     for line in nodetraverseid(nodeid"hlist",head) do
652         for n in nodetraverseid(nodeid"glyph",line.head) do
653             if not(tabularasa_onlytext) or node.has_attribute(n,luatexbase.attributes.tabularasaattr) then
654                 s.kern = n.width
655                 nodeinsertafter(line.list,n,nodecopy(s))
656                 line.head = noderemove(line.list,n)
657             end
658         end
659     end
660     return head
661 end

```

#### 10.17 uppercasecolor

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

```

662 uppercasecolor_onlytext = false
663
664 uppercasecolor = function (head)
665     for line in nodetraverseid(Hhead,head) do
666         for upper in nodetraverseid(GLYPH,line.head) do
667             if not(uppercasecolor_onlytext) or node.has_attribute(upper,luatexbase.attributes.uppercasecolorattr) then
668                 if ((upper.char > 64) and (upper.char < 91)) or

```

```

669         ((upper.char > 57424) and (upper.char < 57451))) then -- for small caps! nice
670         color_push.data = randomcolorstring() -- color or grey string
671         line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
672         nodeinsertafter(line.head,upper,nodecopy(color_pop))
673     end
674 end
675 end
676 end
677 return head
678 end

```

## 10.18 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.

In fact, two boxes are drawn: The first (left) box shows the badness, i. e. the amount of stretching the spaces between words. Too much space results in light grey, 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  $\text{\LaTeX}$ . The box color then corresponds to the amount of font expansion in the line. This works great for demonstrating the positive effect of font expansion on the badness of a line!

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

### 10.18.1 colorstretch – preliminaries

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

```

679 keeptext = true
680 colorexpansion = true
681
682 colorstretch_coloroffset = 0.5
683 colorstretch_colorange = 0.5
684 chickenize_rule_bad_height = 4/5 -- height and depth of the rules
685 chickenize_rule_bad_depth = 1/5
686
687
688 colorstretchnumbers = true
689 drawstretchthreshold = 0.1
690 drawexpansionthreshold = 0.9

```

After these constants have been set, 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 (`colorexansion == 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.

```

691 colorstretch = function (head)
692   local f = font.getfont(font.current()).characters

```

```

693 for line in nodetraverseid(Hhead,head) do
694     local rule_bad = nodenew(RULE)
695
696     if colorexansion then -- if also the font expansion should be shown
697         local g = line.head
698         while not(g.id == 37) do
699             g = g.next
700         end
701         exp_factor = g.width / f[g.char].width
702         exp_color = colorstretch_coloroffset + (1-exp_factor)*10 .. " g"
703         rule_bad.width = 0.5*line.width -- we need two rules on each line!
704     else
705         rule_bad.width = line.width -- only the space expansion should be shown, only one rule
706     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.

```

707     rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bet
708     rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth
709
710     local glue_ratio = 0
711     if line.glue_order == 0 then
712         if line.glue_sign == 1 then
713             glue_ratio = colorstretch_colorange * math.min(line.glue_set,1)
714         else
715             glue_ratio = -colorstretch_colorange * math.min(line.glue_set,1)
716         end
717     end
718     color_push.data = colorstretch_coloroffset + glue_ratio .. " g"
719

```

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

```

720 -- set up output
721     local p = line.head
722
723     -- a rule to immitate kerning all the way back
724     local kern_back = nodenew(RULE)
725     kern_back.width = -line.width
726
727     -- if the text should still be displayed, the color and box nodes are inserted additionally
728     -- and the head is set to the color node
729     if keptext then
730         line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
731     else
732         node.flush_list(p)
733         line.head = nodecopy(color_push)
734     end

```

```

735     nodeinsertafter(line.head,line.head,rule_bad) -- then the rule
736     nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
737     tmpnode = nodeinsertafter(line.head,line.head.next.next,kern_back)
738
739     -- then a rule with the expansion color
740     if colorexansion then -- if also the stretch/shrink of letters should be shown
741         color_push.data = exp_color
742         nodeinsertafter(line.head,tmpnode,nodecopy(color_push))
743         nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
744         nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
745     end

```

Now we are ready with the boxes and stuff and everything. However, a very useful information might be the amount of stretching, not encoded as color, but the real value. In concreto, I mean: narrow boxes get one color, loose boxes get another one, but only if the badness is above a certain amount. This information is printed into the right-hand margin. The threshold is user-adjustable.

```

746     if colorstretchnumbers then
747         j = 1
748         glue_ratio_output = {}
749         for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
750             local char = unicode.utf8.char(s)
751             glue_ratio_output[j] = nodenew(37,1)
752             glue_ratio_output[j].font = font.current()
753             glue_ratio_output[j].char = s
754             j = j+1
755         end
756         if math.abs(glue_ratio) > drawstretchthreshold then
757             if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
758             else color_push.data = "0 0.99 0 rg" end
759         else color_push.data = "0 0 0 rg"
760         end
761
762         nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
763         for i = 1,math.min(j-1,7) do
764             nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])
765         end
766         nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
767     end -- end of stretch number insertion
768 end
769 return head
770 end

```

## dubstepize

FIXME – Isn't that already implemented above? BROOOAR WOBWOBWOB BROOOOAR WOBWOBWOB  
 BROOOOAR WOB WOB WOB ...

771



## scorpionize

This function's intentionally not documented. In memoriam scorpionem. FIXME

```
772 function scorpionize_color(head)
773   color_push.data = ".35 .55 .75 rg"
774   nodeinsertafter(head,head,nodecopy(color_push))
775   nodeinsertafter(head,node.tail(head),nodecopy(color_pop))
776   return head
777 end
```

## 10.19 zebranize

This function is inspired by a discussion with the Heidelberg regular's table and will change the color of each paragraph linewise. Both the textcolor and background color are changed to create a true zebra like look. If you want to change or add colors, just change the values of `zebracolorarray[]` for the text colors and `zebracolorarray_bg[]` for the background. Do not mix with other color changing functions of this package, as that will turn out ugly or erroneous.

The code works just the same as every other thing here: insert color nodes, insert rules, and register the whole thing in `post_linebreak_filter`.

### 10.19.1 zebranize – preliminaries

```
778 zebracolorarray = {}
779 zebracolorarray_bg = {}
780 zebracolorarray[1] = "0.1 g"
781 zebracolorarray[2] = "0.9 g"
782 zebracolorarray_bg[1] = "0.9 g"
783 zebracolorarray_bg[2] = "0.1 g"
```

### 10.19.2 zebranize – the function

This code has to be revisited, it is ugly.

```
784 function zebranize(head)
785   zebracolor = 1
786   for line in nodetraverseid(nodeid"hhead",head) do
787     if zebracolor == #zebracolorarray then zebracolor = 0 end
788     zebracolor = zebracolor + 1
789     color_push.data = zebracolorarray[zebracolor]
790     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
791     for n in nodetraverseid(nodeid"glyph",line.head) do
792       if n.next then else
793         nodeinsertafter(line.head,n,nodecopy(color_pull))
794       end
795     end
796   end
797   local rule_zebra = nodenew(RULE)
798   rule_zebra.width = line.width
```

```

799     rule_zebra.height = tex.baselineskip.width*4/5
800     rule_zebra.depth = tex.baselineskip.width*1/5
801
802     local kern_back = nodenew(RULE)
803     kern_back.width = -line.width
804
805     color_push.data = zebracolorarray_bg[zebracolor]
806     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
807     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
808     nodeinsertafter(line.head,line.head,kern_back)
809     nodeinsertafter(line.head,line.head,rule_zebra)
810 end
811 return (head)
812 end

```

And that's it!



Well, it's not the whole story so far. I plan to test some drawing using only Lua code, writing directly to the pdf file. This section will grow and get better in parallel to my understandings of what's going on. I.e. it will be very slowly ... Nothing here is to be taken as good and/or correct LuaTeXing, and most code is plain ugly. However, it kind of works already ☺

## 11 Drawing

A *very* first, experimental implementation of a drawing of a chicken. The parameters should be consistent, easy to change and that monster should look more like a cute chicken. However, it is chicken, it is Lua, so it belongs into this package. So far, all numbers and positions are hard coded, this will of course change!

```

813 --
814 function pdf_print (...)
815   for _, str in ipairs({...}) do
816     pdf.print(str .. " ")
817   end
818   pdf.print("\string\n")
819 end
820
821 function move (p)
822   pdf_print(p[1],p[2],"m")
823 end
824
825 function line (p)
826   pdf_print(p[1],p[2],"l")
827 end
828
829 function curve(p1,p2,p3)
830   pdf_print(p1[1], p1[2],
831             p2[1], p2[2],
832             p3[1], p3[2], "c")
833 end
834
835 function close ()
836   pdf_print("h")
837 end
838
839 function linewidth (w)
840   pdf_print(w,"w")
841 end
842
843 function stroke ()
844   pdf_print("S")
845 end
846 --
847

```

```

848 function strictcircle(center,radius)
849   local left = {center[1] - radius, center[2]}
850   local lefttop = {left[1], left[2] + 1.45*radius}
851   local leftbot = {left[1], left[2] - 1.45*radius}
852   local right = {center[1] + radius, center[2]}
853   local righttop = {right[1], right[2] + 1.45*radius}
854   local rightbot = {right[1], right[2] - 1.45*radius}
855
856   move (left)
857   curve (lefttop, righttop, right)
858   curve (rightbot, leftbot, left)
859 stroke()
860 end
861
862 function disturb_point(point)
863   return {point[1] + math.random()*5 - 2.5,
864           point[2] + math.random()*5 - 2.5}
865 end
866
867 function sloppycircle(center,radius)
868   local left = disturb_point({center[1] - radius, center[2]})
869   local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
870   local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}
871   local right = disturb_point({center[1] + radius, center[2]})
872   local righttop = disturb_point({right[1], right[2] + 1.45*radius})
873   local rightbot = disturb_point({right[1], right[2] - 1.45*radius})
874
875   local right_end = disturb_point(right)
876
877   move (right)
878   curve (rightbot, leftbot, left)
879   curve (lefttop, righttop, right_end)
880   linewidth(math.random()+0.5)
881   stroke()
882 end
883
884 function sloppyline(start,stop)
885   local start_line = disturb_point(start)
886   local stop_line = disturb_point(stop)
887   start = disturb_point(start)
888   stop = disturb_point(stop)
889   move(start) curve(start_line,stop_line,stop)
890   linewidth(math.random()+0.5)
891   stroke()
892 end

```

## 12 Known Bugs

The behaviour of the `\chickenize` macro is under construction and everything it does so far is considered a feature.

**babel** Using `chickenize` with `babel` leads to a problem with the " (double quote) character, as it is made active: When using `\chickenizesetup` *after* `\begin{document}`, you can *not* use " for strings, but you have to use ' (single quote) instead. No problem really, but take care of this.

## 13 To Dos

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

**rainbowcolor** should be more flexible – the angle of the rainbow should be easily adjustable.

**pancakenize** should do something funny.

**chickenize** should differ between character and punctuation.

**swing** swing dancing apes – that will be very hard, actually ...

**chickenmath** chickenization of math mode

## 14 Literature

The following list directs you to helpful literature that will help you to better understand the concepts used in this package and for in-depth explanation. Also, most of the code here is taken from or based on this literature, so it is also a list of references somehow:

- Lua<sub>T</sub><sub>E</sub>X documentation – the manual and links to presentations and talks: <http://www.luatex.org/documentation.html>
- The Lua manual, for Lua 5.1: <http://www.lua.org/manual/5.1/>
- Programming in Lua, 1<sup>st</sup> edition, aiming at Lua 5.0, but still (largely) valid for 5.1: <http://www.lua.org/pil/>

## 15 Thanks

This package would not have been possible without the help of many people who patiently answered my annoying questions on mailing lists and in personal mails. And of course not without the work of the Lua<sub>T</sub><sub>E</sub>X team!

Special thanks go to Paul “we could have chickenized the world” Isambert who contributed a lot of ideas, code and bug fixes and made much of the code executable at all. I also think Philipp Gesang who gave me many advices on the Lua code – which I still didn't have time to correct ...