

*»The Monty Pythons, were they T<sub>E</sub>X users,  
could have written the chickenize macro.«*

Paul Isambert

# CHICKENIZE

Arno Trautmann

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

This is the package `chickenize`. It allows manipulations of any LuaT<sub>E</sub>X 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 T<sub>E</sub>X 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.

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

---

<sup>1</sup>The code is based on pure LuaT<sub>E</sub>X features, so don't even try to use it with any other T<sub>E</sub>X flavour. The package is tested under plain LuaT<sub>E</sub>X and Lua<sup>La</sup>T<sub>E</sub>X. If you tried using it with ConT<sub>E</sub>Xt, please share your experience, I will gladly try to make it compatible!

---

### maybe useful functions

---

<a href="#">colorstretch</a>	shows grey boxes that depict 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>	changes 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
<a href="#">matrixize</a>	replaces every glyph by its ASCII value in binary code
<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>
<b>1</b>	<b>How It Works</b>	<b>5</b>
<b>2</b>	<b>Commands – How You Can Use It</b>	<b>5</b>
2.1	TeX Commands – Document Wide . . . . .	5
2.2	How to Deactivate It . . . . .	6
2.3	\text-Versions . . . . .	7
2.4	Lua functions . . . . .	7
<b>3</b>	<b>Options – How to Adjust It</b>	<b>7</b>
<b>II</b>	<b>Tutorial</b>	<b>10</b>
<b>4</b>	<b>Lua code</b>	<b>10</b>
<b>5</b>	<b>callbacks</b>	<b>10</b>
5.1	How to use a callback . . . . .	11
<b>6</b>	<b>Nodes</b>	<b>11</b>
<b>7</b>	<b>Other things</b>	<b>12</b>
<b>III</b>	<b>Implementation</b>	<b>14</b>
<b>8</b>	<b>TeX file</b>	<b>14</b>
<b>9</b>	<b>LaTeX package</b>	<b>19</b>
9.1	Definition of User-Level Macros . . . . .	19
<b>10</b>	<b>Lua Module</b>	<b>20</b>
10.1	chickenize . . . . .	20
10.2	guttenbergenize . . . . .	23
10.2.1	guttenbergenize – preliminaries . . . . .	23
10.2.2	guttenbergenize – the function . . . . .	23
10.3	hammertime . . . . .	24
10.4	itsame . . . . .	24
10.5	kernmanipulate . . . . .	25

10.6	leetspeak	26
10.7	letterspaceadjust	26
10.7.1	setup of variables	27
10.7.2	function implementation	27
10.8	matrixize	27
10.9	pancakenize	28
10.10	randomfonts	28
10.11	randomucl	29
10.12	randomchars	29
10.13	randomcolor and rainbowcolor	29
10.13.1	randomcolor – preliminaries	29
10.13.2	randomcolor – the function	31
10.14	rickroll	31
10.15	tabularasa	31
10.16	uppercasecolor	32
10.17	colorstretch	32
10.17.1	colorstretch – preliminaries	32
10.18	zebranize	35
10.18.1	zebranize – preliminaries	35
10.18.2	zebranize – the function	36
<b>11</b>	<b>Drawing</b>	<b>37</b>
<b>12</b>	<b>Known Bugs</b>	<b>40</b>
<b>13</b>	<b>To Dos</b>	<b>40</b>
<b>14</b>	<b>Literature</b>	<b>40</b>
<b>15</b>	<b>Thanks</b>	<b>40</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 (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.

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 this package – you can either stay on the  $\TeX$  side or use the Lua functions directly. In fact, the  $\TeX$  macros are simple wrappers around the functions.

#### 2.1 $\TeX$ Commands – Document Wide

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

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

**`\chickenize`** Replaces every word of the input with the word “chicken”. Maybe sometime the replaced word can be changed, but up to now, it's only chicken. To be a bit less static, about every 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 WOBBBWOB BWOB BZZZR-RRRRRRROOOOOOAAAAA ... (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 ...

---

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

`\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.

`\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 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.

`\pancakelize` This is a dummy so far, as I have no idea what it should do. If you have suggestions, please tell me.

`\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 sequence representing 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 box greyness give you information about how well the overall greyness of the typeset page is.

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

If you want to manipulate only a part of a paragraph, you have to 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.

### 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 have<sup>4</sup> 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.<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 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 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 *care-*

---

<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.

<sup>5</sup>On a 500 pages text-only  $\text{\LaTeX}$  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`!

*ful!* 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  $\text{\TeX}$  side meaning that you can use *only* % as comment string. If you use `--`, all of the argument will be ignored as  $\text{\TeX}$  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 keep kind of track of the options and variables. There is no guarantee for this list, and if you find something that is missing or doesn't work as described here, please inform me!

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

`chickenstring = <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`.)

`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 = <bool>` 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`).

`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 `\randomucl` 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.

`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 for learning purposes. However, this is *not* intended as a comprehensive LuaTeX tutorial. It's just to get an idea how things work here. For a deeper understanding of LuaTeX you should consult the LuaTeX manual and also some Lua introduction like “Programming in Lua”.

## 4 Lua code

The crucial new thing in LuaTeX 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 can be used for 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 make the thing useful for TeXing, especially the `tex.` library that offers access to TeX. In the simple example above, the function `tex.print()` inserts its argument into the TeX input stream, so the result of the calculation (10) is printed in the document.

Larger parts of Lua code should not be in the same file as your TeX code, but rather in a separate file. That can then be loaded using

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

If you use Lua<sup>A</sup>TeX, 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 TeX behaves: The callbacks. A callback is a point where you can hook into TeX's working and do anything that may make sense – or not. (Thus maybe breaking your document completely ...)

Callbacks are used at several points of TeX'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.)  $\text{\TeX}$  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  $\text{\TeX}$ '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 we don't use this syntax here, but make use of 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 offers a possibility to remove functions from callbacks, and then you need a unique name for the function:

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

You have to consult the  $\text{\LaTeX}$  manual to see what functionality a callback has, when it is 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  $\text{\LaTeX}$  manual and the `luatexbase` documentation for details!

## 6 Nodes

Essentially everything that  $\text{\LaTeX}$  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 go through 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. For this, the function `node.traverse_id(37,head)` can be used, 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. That makes no sense, but it is a good example:

```
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 LuaTeX manual! Then, you have to remove the `if n.char` line as glue nodes don't have a `.char`. 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 go through a whole vertical stack of horizontal lists, vertical glue and other material. See some of the functions below to understand what is necessary then.

## 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 use synonyms like `nodetraverseid = node.traverse_id` instead of the original names.

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

The namespace of this package is *not* consistent. Please don't take anything here as an example for good Lua coding, for good TeXing or even for good LuaTeXing. It's not. For

really good code, check out the code written by Hans Hagen or other professionals. If you understand this package here, 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 T<sub>E</sub>X file

This file is more-or-less just a dummy file to offer a nice interface for the functions. Basically, every macro registers the function with the same name in the corresponding callback. The un-macros remove the functions. If it makes sense, there are text-variants that activate the function only in a certain area of the text, using LuaT<sub>E</sub>X's attributes.

For (un)registering, we use the `luatexbase` package. Then, the `.lua` file is loaded which does the actual work. Finally, the T<sub>E</sub>X macros are defined as simple `\directlua` calls.

```
1 \input{luatexbase.sty}
2 \directlua{dofile("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] = "BZZRRRRRRRROOOOOOAAAAA"
51
52     chickenizefraction = 1
53   }
54 }
55 \let\dubstepize\dubstepenize
56
57 \def\guttenbergenize{ %% 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",gutenbergize_rq,"gutenbergize_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{      %% to be implemented
117   \directlua{}}
118 \def\unmilkcow{
119   \directlua{}}
120
121 \def\pancakenize{      %% to be implemented
122   \directlua{}}
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\randomfonts{
141   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomfonts,"randomfonts")}}
142 \def\unrandomfonts{
143   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomfonts")}}
144
145 \def\randomuclc{
146   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",randomuclc,"randomuclc")}}
147 \def\unrandomuclc{
148   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","randomuclc")}}
149
150 \def\scorpionize{
151   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",scorpionize_color,"scorpionize_color")}}
152 \def\unscorpionize{
153   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","scorpionize_color")}}
154
155 \def\spankmonkey{ %% to be implemented
156   \directlua{}}
157 \def\unspankmonkey{
158   \directlua{}}
159
160 \def\tabularasa{
161   \directlua{luatexbase.add_to_callback("post_linebreak_filter",tabularasa,"tabularasa")}}
162 \def\untabularasa{
163   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","tabularasa")}}
164
165 \def\uppercasecolor{
166   \directlua{luatexbase.add_to_callback("post_linebreak_filter",uppercasecolor,"uppercasecolor")}}
167 \def\unuppercasecolor{
168   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","uppercasecolor")}}
169
170 \def\zebranize{

```

```

171 \directlua{luatexbase.add_to_callback("post_linebreak_filter",zebranize,"zebranize")}}
172 \def\unzebranize{
173 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","zebranize")}}

```

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

```

174 \newluatexattribute\leetattr
175 \newluatexattribute\randcolorattr
176 \newluatexattribute\randfontsaattr
177 \newluatexattribute\randuclcatr
178 \newluatexattribute\tabularasaattr
179
180 \long\def\textleetspeak#1%
181 {\setluatexattribute\leetattr{42}#1\unsetluatexattribute\leetattr}
182 \long\def\textrandomcolor#1%
183 {\setluatexattribute\randcolorattr{42}#1\unsetluatexattribute\randcolorattr}
184 \long\def\textrandomfontsa#1%
185 {\setluatexattribute\randfontsaattr{42}#1\unsetluatexattribute\randfontsaattr}
186 \long\def\textrandomfontsa#1%
187 {\setluatexattribute\randfontsaattr{42}#1\unsetluatexattribute\randfontsaattr}
188 \long\def\textrandomuclc#1%
189 {\setluatexattribute\randuclcatr{42}#1\unsetluatexattribute\randuclcatr}
190 \long\def\texttabularasa#1%
191 {\setluatexattribute\tabularasaattr{42}#1\unsetluatexattribute\tabularasaattr}

```

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.

```

192 \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.

```

193 \long\def\luadraw#1#2{%
194 \vbox to #1bp{%
195 \vfil
196 \luatexlatalua{pdf_print("q") #2 pdf_print("Q")}%
197 }%
198 }
199 \long\def\drawchicken{
200 \luadraw{90}{
201 kopf = {200,50} % Kopfmitte
202 kopf_rad = 20
203
204 d = {215,35} % Halsansatz
205 e = {230,10} %
206
207 korper = {260,-10}
208 korper_rad = 40

```

```

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

```

## 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 want to use anything of the features presented here, you have to load the packages on your own. Maybe this will change.

```

235 \ProvidesPackage{chickenize}%
236 [2011/10/22 v0.1 chickenize package]
237 \input{chickenize}

```

### 9.1 Definition of User-Level Macros

```

238 %% We want to "chickenize" figures, too. So ...

```

```

239 \iffalse
240 \DeclareDocumentCommand\includegraphics{0{m}}{
241     \fbox{Chicken} %% actually, I'd love to draw an MP graph showing a chicken ...
242 }
243 %%% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
244 %% So far, you have to load pgfplots yourself.
245 %% As it is a mighty package, I don't want the user to force loading it.
246 \NewDocumentCommand\balmerpeak{G{}0{-4cm}}{
247 %% to be done using Lua drawing.
248 }
249 \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 on document level, too.

```

250
251 local nodenew = node.new
252 local nodecopy = node.copy
253 local nodeinsertbefore = node.insert_before
254 local nodeinsertafter = node.insert_after
255 local noderemove = node.remove
256 local nodeid = node.id
257 local nodetraverseid = node.traverse_id
258
259 Hhead = nodeid("hhead")
260 RULE = nodeid("rule")
261 GLUE = nodeid("glue")
262 WHAT = nodeid("whatsit")
263 COL = node.subtype("pdf_colorstack")
264 GLYPH = nodeid("glyph")

```

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

```

265 color_push = nodenew(WHAT,COL)
266 color_pop = nodenew(WHAT,COL)
267 color_push.stack = 0
268 color_pop.stack = 0
269 color_push.cmd = 1
270 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.

```
271 chicken_pagenumbers = true
272
273 chickenstring = {}
274 chickenstring[1] = "Chicken" -- chickenstring is a table, please remeber this!
275
276 chickenizefraction = 0.5
277 -- set this to a small value to fool somebody, or to see if your text has been read carefully. Th
278 chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing your
279
280 local tbl = font.getfont(font.current())
281 local space = tbl.parameters.space
282 local shrink = tbl.parameters.space_shrink
283 local stretch = tbl.parameters.space_stretch
284 local match = unicode.utf8.match
285 chickenize_ignore_word = false
286
287 chickenize_real_stuff = function(i,head)
288     while ((i.next.id == 37) or (i.next.id == 11) or (i.next.id == 7) or (i.next.id == 0)) do --
289         i.next = i.next.next
290     end
291
292     chicken = {} -- constructing the node list.
293
294 -- Should this be done only once? No, then we loose the freedom to change the string in-document.
295 -- But it could be done only once each paragraph as in-paragraph changes are not possible!
296
297     chickenstring_tmp = chickenstring[math.random(1,#chickenstring)]
298     chicken[0] = nodenew(37,1) -- only a dummy for the loop
299     for i = 1,string.len(chickenstring_tmp) do
300         chicken[i] = nodenew(37,1)
301         chicken[i].font = font.current()
302         chicken[i-1].next = chicken[i]
303     end
304
305     j = 1
306     for s in string.utfvalues(chickenstring_tmp) do
307         local char = unicode.utf8.char(s)
308         chicken[j].char = s
309         if match(char,"%s") then
310             chicken[j] = nodenew(10)
311             chicken[j].spec = nodenew(47)
312             chicken[j].spec.width = space
313             chicken[j].spec.shrink = shrink
314             chicken[j].spec.stretch = stretch
315         end
316     end
```

```

316     j = j+1
317 end
318
319 node.slide(chicken[1])
320 lang.hyphenate(chicken[1])
321 chicken[1] = node.kerning(chicken[1])    -- FIXME: does not work
322 chicken[1] = node.ligaturing(chicken[1]) -- dito
323
324 nodeinsertbefore(head,i,chicken[1])
325 chicken[1].next = chicken[2] -- seems to be necessary ... to be fixed
326 chicken[string.len(chickenstring_tmp)].next = i.next
327 return head
328 end
329
330 chickenize = function(head)
331   for i in nodetraverseid(37,head) do --find start of a word
332     if (chickenize_ignore_word == false) then -- normal case: at the beginning of a word, we jump
333       head = chickenize_real_stuff(i,head)
334     end
335
336 -- At the end of the word, the ignoring is reset. New chance for everyone.
337     if not((i.next.id == 37) or (i.next.id == 7) or (i.next.id == 22) or (i.next.id == 11)) then
338       chickenize_ignore_word = false
339     end
340
341 -- And the random determination of the chickenization of the next word:
342     if math.random() > chickenizefraction then
343       chickenize_ignore_word = true
344     else if chickencount then
345       chicken_substitutions = chicken_substitutions + 1
346     end
347   end
348 end
349 return head
350 end
351
352 local separator = string.rep("=", 28)
353 local texiowrite_nl = texio.write_nl
354 nicetext = function()
355   texiowrite_nl("Output written on "..tex.jobname.."pdf ("..status.total_pages.." chicken,".." eg
356   texiowrite_nl(" ")
357   texiowrite_nl(separator)
358   texiowrite_nl("Hello my dear user,")
359   texiowrite_nl("good job, now go outside and enjoy the world!")
360   texiowrite_nl(" ")
361   texiowrite_nl("And don't forget to feed your chicken!")

```

```

362 texiowrite_nl(separator .. "\n")
363 if chickencount then
364     texiowrite_nl("There were "..chicken_substitutions.." substitutions made.")
365     texiowrite_nl(separator)
366 end
367 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 way Lua offers for our needs. Learn it, this might be helpful for you sometime, too.

```

368 local quotestrings = {[171] = true, [172] = true,
369   [8216] = true, [8217] = true, [8218] = true,
370   [8219] = true, [8220] = true, [8221] = true,
371   [8222] = true, [8223] = true,
372   [8248] = true, [8249] = true, [8250] = true}

```

### 10.2.2 guttenbergenize – the function

```

373 guttenbergenize_rq = function(head)
374   for n in nodetraverseid(nodeid"glyph",head) do
375     local i = n.char
376     if quotestrings[i] then
377       noderemove(head,n)
378     end
379   end
380   return head
381 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>

```
382 hammertimedelay = 1.2
383 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
384 hammertime = function(head)
385   if hammerfirst then
386     texiowrite_nl(htime_separator)
387     texiowrite_nl("=====STOP!=====\\n")
388     texiowrite_nl(htime_separator .. "\\n\\n\\n")
389     os.sleep (hammertimedelay*1.5)
390     texiowrite_nl(htime_separator .. "\\n")
391     texiowrite_nl("=====HAMMERTIME=====\\n")
392     texiowrite_nl(htime_separator .. "\\n\\n")
393     os.sleep (hammertimedelay)
394     hammerfirst = false
395   else
396     os.sleep (hammertimedelay)
397     texiowrite_nl(htime_separator)
398     texiowrite_nl("=====U can't touch this!=====\\n")
399     texiowrite_nl(htime_separator .. "\\n\\n")
400     os.sleep (hammertimedelay*0.5)
401   end
402   return head
403 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.

```
404 itsame = function()
405 local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
406 color = "1 .6 0"
407 for i = 6,9 do mr(i,3) end
408 for i = 3,11 do mr(i,4) end
409 for i = 3,12 do mr(i,5) end
410 for i = 4,8 do mr(i,6) end
411 for i = 4,10 do mr(i,7) end
412 for i = 1,12 do mr(i,11) end
413 for i = 1,12 do mr(i,12) end
```

---

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



```

414 for i = 1,12 do mr(i,13) end
415
416 color = ".3 .5 .2"
417 for i = 3,5 do mr(i,3) end mr(8,3)
418 mr(2,4) mr(4,4) mr(8,4)
419 mr(2,5) mr(4,5) mr(5,5) mr(9,5)
420 mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
421 for i = 3,8 do mr(i,8) end
422 for i = 2,11 do mr(i,9) end
423 for i = 1,12 do mr(i,10) end
424 mr(3,11) mr(10,11)
425 for i = 2,4 do mr(i,15) end for i = 9,11 do mr(i,15) end
426 for i = 1,4 do mr(i,16) end for i = 9,12 do mr(i,16) end
427
428 color = "1 0 0"
429 for i = 4,9 do mr(i,1) end
430 for i = 3,12 do mr(i,2) end
431 for i = 8,10 do mr(5,i) end
432 for i = 5,8 do mr(i,10) end
433 mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
434 for i = 4,9 do mr(i,12) end
435 for i = 3,10 do mr(i,13) end
436 for i = 3,5 do mr(i,14) end
437 for i = 7,10 do mr(i,14) end
438 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.

```

439 chickenkernamount = 0
440 chickeninvertkerning = false
441
442 function kernmanipulate (head)
443   if chickeninvertkerning then -- invert the kerning
444     for n in nodetraverseid(11,head) do
445       n.kern = -n.kern
446     end
447   else -- if not, set it to the given value
448     for n in nodetraverseid(11,head) do
449       n.kern = chickenkernamount
450     end

```

```

451 end
452 return head
453 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.

```

454 leetspeak_onlytext = false
455 leetable = {
456   [101] = 51, -- E
457   [105] = 49, -- I
458   [108] = 49, -- L
459   [111] = 48, -- O
460   [115] = 53, -- S
461   [116] = 55, -- T
462
463   [101-32] = 51, -- e
464   [105-32] = 49, -- i
465   [108-32] = 49, -- l
466   [111-32] = 48, -- o
467   [115-32] = 53, -- s
468   [116-32] = 55, -- t
469 }

```

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

```

470 leet = function(head)
471   for line in nodetraverseid(Hhead,head) do
472     for i in nodetraverseid(GLYPH,line.head) do
473       if not leetspeak_onlytext or
474         node.has_attribute(i,luatexbase.attributes.leetattr)
475       then
476         if leetable[i.char] then
477           i.char = leetable[i.char]
478         end
479       end
480     end
481   end
482   return head
483 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 and the greyness of a page (hopefully) comes out more equally.

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

```
484 local letterspace_glue = nodenew(nodeid"glue")
485 local letterspace_spec = nodenew(nodeid"glue_spec")
486 local letterspace_pen = nodenew(nodeid"penalty")
487
488 letterspace_spec.width = tex.sp"0pt"
489 letterspace_spec.stretch = tex.sp"2pt"
490 letterspace_glue.spec = letterspace_spec
491 letterspace_pen.penalty = 10000
```

### 10.7.2 function implementation

```
492 letterspaceadjust = function(head)
493   for glyph in nodetraverseid(nodeid"glyph", head) do
494     if glyph.prev and (glyph.prev.id == nodeid"glyph" or glyph.prev.id == nodeid"disc") then
495       local g = nodecopy(letterspace_glue)
496       nodeinsertbefore(head, glyph, g)
497       nodeinsertbefore(head, g, nodecopy(letterspace_pen))
498     end
499   end
500   return head
501 end
```

## 10.8 matrixize

Substitutes every glyph by a representation of its ASCII value. Might be extended to cover full unicode, 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.

```
502 matrixize = function(head)
503   x = {}
504   s = nodenew(nodeid"disc")
505   for n in nodetraverseid(nodeid"glyph", head) do
506     j = n.char
507     for m = 0,7 do -- stay ASCII for now
508       x[7-m] = nodecopy(n) -- to get the same font etc.
509     end
510     if (j / (2^(7-m)) < 1) then
511       x[7-m].char = 48
512     else
```

```

513         x[7-m].char = 49
514         j = j-(2^(7-m))
515     end
516     nodeinsertbefore(head,n,x[7-m])
517     nodeinsertafter(head,x[7-m],nodecopy(s))
518 end
519 noderemove(head,n)
520 end
521 return head
522 end

```

## 10.9 pancakenize

Not yet completely decided what this should do, but it might come down to inserting a cooking recipe for a ... well, guess what. Possible implementations are: Substitute a whole sentence, from full-stop to full-stop. OR: Substitute word-by-word at a random place. OR (expert-freak-1337-level): Substitute the n-th word of each page to a word of the recipe. That would be totally awesome!!

## 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.

```

523 local randomfontslower = 1
524 local randomfontsupper = 0
525 %
526 randomfonts = function(head)
527     local rfub
528     if randomfontsupper > 0 then -- fixme: this should be done only once, no? Or at every paragraph
529         rfub = randomfontsupper -- user-specified value
530     else
531         rfub = font.max() -- or just take all fonts
532     end
533     for line in nodetraverseid(Hhead,head) do
534         for i in nodetraverseid(GLYPH,line.head) do
535             if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattrib) then
536                 i.font = math.random(randomfontslower,rfub)
537             end
538         end
539     end
540     return head
541 end

```

## 10.11 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```
542 uclcratio = 0.5 -- ratio between uppercase and lower case
543 randomuclc = function(head)
544   for i in nodetraverseid(37,head) do
545     if not(randomuclc_onlytext) or node.has_attribute(i,luatexbase.attributes.randuclcatr) then
546       if math.random() < uclcratio then
547         i.char = tex.uccode[i.char]
548       else
549         i.char = tex.lccode[i.char]
550       end
551     end
552   end
553   return head
554 end
```

## 10.12 randomchars

```
555 randomchars = function(head)
556   for line in nodetraverseid(Hhead,head) do
557     for i in nodetraverseid(GLYPH,line.head) do
558       i.char = math.floor(math.random()*512)
559     end
560   end
561   return head
562 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.

```
563 randomcolor_grey = false
564 randomcolor_onlytext = false --switch between local and global colorization
565 rainbowcolor = false
566
567 grey_lower = 0
568 grey_upper = 900
569
570 Rgb_lower = 1
571 rGb_lower = 1
572 rgB_lower = 1
573 Rgb_upper = 254
```

```

574 rGb_upper = 254
575 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.

```

576 rainbow_step = 0.005
577 rainbow_Rgb = 1-rainbow_step -- we start in the red phase
578 rainbow_rGb = rainbow_step -- values x must always be 0 < x < 1
579 rainbow_rgB = rainbow_step
580 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.

```

581 randomcolorstring = function()
582   if randomcolor_grey then
583     return (0.001*math.random(grey_lower, grey_upper)).." g"
584   elseif rainbowcolor then
585     if rainind == 1 then -- red
586       rainbow_rGb = rainbow_rGb + rainbow_step
587       if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
588     elseif rainind == 2 then -- yellow
589       rainbow_Rgb = rainbow_Rgb - rainbow_step
590       if rainbow_Rgb <= rainbow_step then rainind = 3 end
591     elseif rainind == 3 then -- green
592       rainbow_rgB = rainbow_rgB + rainbow_step
593       rainbow_rGb = rainbow_rGb - rainbow_step
594       if rainbow_rGb <= rainbow_step then rainind = 4 end
595     elseif rainind == 4 then -- blue
596       rainbow_Rgb = rainbow_Rgb + rainbow_step
597       if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
598     else -- purple
599       rainbow_rgB = rainbow_rgB - rainbow_step
600       if rainbow_rgB <= rainbow_step then rainind = 1 end
601     end
602     return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rgB.." rg"
603   else
604     Rgb = math.random(Rgb_lower, Rgb_upper)/255
605     rGb = math.random(rGb_lower, rGb_upper)/255
606     rgB = math.random(rgB_lower, rgB_upper)/255
607     return Rgb.." "..rGb.." "..rgB.." "..." rg"
608   end
609 end

```

### 10.13.2 randomcolor – the function

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

```
610 randomcolor = function(head)
611   for line in nodetraverseid(0,head) do
612     for i in nodetraverseid(37,line.head) do
613       if not(randomcolor_onlytext) or
614         (node.has_attribute(i,luatexbase.attributes.randcolorattr))
615       then
616         color_push.data = randomcolorstring() -- color or grey string
617         line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
618         nodeinsertafter(line.head,i,nodecopy(color_pop))
619       end
620     end
621   end
622   return head
623 end
```

### 10.14 rickroll

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

### 10.15 tabularasa

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

```
624 tabularasa_onlytext = false
625
626 tabularasa = function(head)
627   local s = nodenew(nodeid"kern")
628   for line in nodetraverseid(nodeid"hlist",head) do
629     for n in nodetraverseid(nodeid"glyph",line.list) do
630       if not(tabularasa_onlytext) or node.has_attribute(n,luatexbase.attributes.tabularasaattr) then
631         s.kern = n.width
632         nodeinsertafter(line.list,n,nodecopy(s))
633         line.head = noderemove(line.list,n)
634       end
635     end
636   end
637   return head
638 end
```

## 10.16 uppercasecolor

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

```
639 uppercasecolor = function (head)
640   for line in nodetraverseid(Hhead,head) do
641     for upper in nodetraverseid(GLYPH,line.head) do
642       if (((upper.char > 64) and (upper.char < 91)) or
643         ((upper.char > 57424) and (upper.char < 57451))) then -- for small caps! nice
644         color_push.data = randomcolorstring() -- color or grey string
645         line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
646         nodeinsertafter(line.head,upper,nodecopy(color_pop))
647       end
648     end
649   end
650   return head
651 end
```

## 10.17 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 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!

### 10.17.1 colorstretch – preliminaries

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

```
652 keeptext = true
653 colorexpansion = true
654
655 colorstretch_coloroffset = 0.5
656 colorstretch_colorange = 0.5
657 chickenize_rule_bad_height = 4/5 -- height and depth of the rules
658 chickenize_rule_bad_depth = 1/5
```



```

659
660
661 colorstretchnumbers = true
662 drawstretchthreshold = 0.1
663 drawexpansionthreshold = 0.9

```

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 (colorexpan == 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.

```

664 colorstretch = function (head)
665   local f = font.getfont(font.current()).characters
666   for line in nodetraverseid(Hhead,head) do
667     local rule_bad = nodenew(RULE)
668
669     if colorexpan then -- if also the font expansion should be shown
670       local g = line.head
671       while not(g.id == 37) do
672         g = g.next
673       end
674       exp_factor = g.width / f[g.char].width
675       exp_color = colorstretch_coloroffset + (1-exp_factor)*10 .. " g"
676       rule_bad.width = 0.5*line.width -- we need two rules on each line!
677     else
678       rule_bad.width = line.width -- only the space expansion should be shown, only one rule
679     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.

```

680   rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bet
681   rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth
682
683   local glue_ratio = 0
684   if line.glue_order == 0 then
685     if line.glue_sign == 1 then
686       glue_ratio = colorstretch_colorange * math.min(line.glue_set,1)
687     else
688       glue_ratio = -colorstretch_colorange * math.min(line.glue_set,1)
689     end
690   end
691   color_push.data = colorstretch_coloroffset + glue_ratio .. " g"
692

```

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

```

693 -- set up output
694     local p = line.head
695
696 -- a rule to immitate kerning all the way back
697     local kern_back = nodenew(RULE)
698     kern_back.width = -line.width
699
700 -- if the text should still be displayed, the color and box nodes are inserted additionally
701 -- and the head is set to the color node
702     if keeptext then
703         line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
704     else
705         node.flush_list(p)
706         line.head = nodecopy(color_push)
707     end
708     nodeinsertafter(line.head,line.head,rule_bad) -- then the rule
709     nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
710     tmpnode = nodeinsertafter(line.head,line.head.next.next,kern_back)
711
712 -- then a rule with the expansion color
713 if colorexansion then -- if also the stretch/shrink of letters should be shown
714     color_push.data = exp_color
715     nodeinsertafter(line.head,tmpnode,nodecopy(color_push))
716     nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
717     nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
718 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.

```

719     if colorstretchnumbers then
720         j = 1
721         glue_ratio_output = {}
722         for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
723             local char = unicode.utf8.char(s)
724             glue_ratio_output[j] = nodenew(37,1)
725             glue_ratio_output[j].font = font.current()
726             glue_ratio_output[j].char = s
727             j = j+1
728         end
729         if math.abs(glue_ratio) > drawstretchthreshold then
730             if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
731             else color_push.data = "0 0.99 0 rg" end
732         else color_push.data = "0 0 0 rg"

```

```

733     end
734
735     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
736     for i = 1,math.min(j-1,7) do
737         nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])
738     end
739     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
740 end -- end of stretch number insertion
741 end
742 return head
743 end

```

## dubstepize

BROOOAR WOBWOBWOB BROOOOAR WOBWOBWOB BROOOOAR WOB WOB WOB  
 ...  
 744

## scorpionize

These functions intentionally not documented.

```

745 function scorpionize_color(head)
746     color_push.data = ".35 .55 .75 rg"
747     nodeinsertafter(head,head,nodecopy(color_push))
748     nodeinsertafter(head,node.tail(head),nodecopy(color_pop))
749     return head
750 end

```

## 10.18 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`.

### 10.18.1 zebranize – preliminaries

```

751 zebracolorarray = {}
752 zebracolorarray_bg = {}
753 zebracolorarray[1] = "0.1 g"

```

```

754 zebracolorarray[2] = "0.9 g"
755 zebracolorarray_bg[1] = "0.9 g"
756 zebracolorarray_bg[2] = "0.1 g"

```

### 10.18.2 zebranize – the function

This code has to be revisited, it is ugly.

```

757 function zebranize(head)
758   zebracolor = 1
759   for line in nodetraverseid(nodeid"hhead",head) do
760     if zebracolor == #zebracolorarray then zebracolor = 0 end
761     zebracolor = zebracolor + 1
762     color_push.data = zebracolorarray[zebracolor]
763     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
764     for n in nodetraverseid(nodeid"glyph",line.head) do
765       if n.next then else
766         nodeinsertafter(line.head,n,nodecopy(color_pull))
767       end
768     end
769
770     local rule_zebra = nodenew(RULE)
771     rule_zebra.width = line.width
772     rule_zebra.height = tex.baselineskip.width*4/5
773     rule_zebra.depth = tex.baselineskip.width*1/5
774
775     local kern_back = nodenew(RULE)
776     kern_back.width = -line.width
777
778     color_push.data = zebracolorarray_bg[zebracolor]
779     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
780     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
781     nodeinsertafter(line.head,line.head,kern_back)
782     nodeinsertafter(line.head,line.head,rule_zebra)
783   end
784   return (head)
785 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!

```
786 --
787 function pdf_print (...)
788   for _, str in ipairs({...}) do
789     pdf.print(str .. " ")
790   end
791   pdf.print("\string\n")
792 end
793
794 function move (p)
795   pdf_print(p[1],p[2],"m")
796 end
797
798 function line (p)
799   pdf_print(p[1],p[2],"l")
800 end
801
802 function curve(p1,p2,p3)
803   pdf_print(p1[1], p1[2],
804             p2[1], p2[2],
805             p3[1], p3[2], "c")
806 end
807
808 function close ()
809   pdf_print("h")
810 end
811
812 function linewidth (w)
813   pdf_print(w,"w")
814 end
815
816 function stroke ()
817   pdf_print("S")
```

```

818 end
819 --
820
821 function strictcircle(center,radius)
822   local left = {center[1] - radius, center[2]}
823   local lefttop = {left[1], left[2] + 1.45*radius}
824   local leftbot = {left[1], left[2] - 1.45*radius}
825   local right = {center[1] + radius, center[2]}
826   local righttop = {right[1], right[2] + 1.45*radius}
827   local rightbot = {right[1], right[2] - 1.45*radius}
828
829   move (left)
830   curve (lefttop, righttop, right)
831   curve (rightbot, leftbot, left)
832 stroke()
833 end
834
835 function disturb_point(point)
836   return {point[1] + math.random()*5 - 2.5,
837           point[2] + math.random()*5 - 2.5}
838 end
839
840 function sloppycircle(center,radius)
841   local left = disturb_point({center[1] - radius, center[2]})
842   local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
843   local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}
844   local right = disturb_point({center[1] + radius, center[2]})
845   local righttop = disturb_point({right[1], right[2] + 1.45*radius})
846   local rightbot = disturb_point({right[1], right[2] - 1.45*radius})
847
848   local right_end = disturb_point(right)
849
850   move (right)
851   curve (rightbot, leftbot, left)
852   curve (lefttop, righttop, right_end)
853   linewidth(math.random()+0.5)
854   stroke()
855 end
856
857 function sloppyline(start,stop)
858   local start_line = disturb_point(start)
859   local stop_line = disturb_point(stop)
860   start = disturb_point(start)
861   stop = disturb_point(stop)
862   move(start) curve(start_line,stop_line,stop)
863   linewidth(math.random()+0.5)

```

```
864 stroke()  
865 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 `"` character, as it is made active: When using `\chickenizesetup` *after* `\begin{document}`, you can *not* use `"` for strings, but you have to use `'`. 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:

- LuaTeX 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 LuaTeX 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.