

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

Paul Isambert

CHICKENIZE

v0.1

Arno Trautmann

arno.trautmann@gmx.de

This is the documentation of the package `chickenize`. It allows manipulations of any LuaT_EX document¹ 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_EX 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'.

¹The code is based on pure LuaT_EX features, so don't even try to use it with any other T_EX flavour. The package is tested under plain LuaT_EX and Lua^{La}T_EX. If you tried using it with ConT_EXt, please share your experience, I will gladly try to make it compatible!

maybe useful functions

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

less useful functions

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

complete nonsense

chickenize	replaces every word with “chicken”
gutenbergize	deletes every quote and footnotes
hammertime	U can't touch this!
kernmanipulation	manipulates the kerning
matrixize	replaces every glyph by its ASCII value in binary code
randomfonts	changes the font randomly between every letter
randomchars	randomizes the (letters of the) whole input

Contents

I	User Documentation	5
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
II	Tutorial	10
4	Lua code	10
5	callbacks	10
5.1	How to use a callback	11
6	Nodes	11
7	Other things	12
III	Implementation	13
8	TeX file	13
9	LaTeX package	18
9.1	Definition of User-Level Macros	18
10	Lua Module	19
10.1	chickenize	19
10.2	guttenbergenize	22
10.2.1	guttenbergenize – preliminaries	22
10.2.2	guttenbergenize – the function	22
10.3	hammertime	22
10.4	itsame	23
10.5	kernmanipulate	24
10.6	leetspeak	24
10.7	letterspaceadjust	25
10.7.1	setup of variables	25
10.7.2	function implementation	26
10.8	matrixize	26

10.9	pancakenize	26
10.10	randomfonts	27
10.11	randomucl	27
10.12	randomchars	27
10.13	randomcolor and rainbowcolor	28
10.13.1	randomcolor – preliminaries	28
10.13.2	randomcolor – the function	29
10.14	rickroll	29
10.15	tabularasa	30
10.16	uppercasecolor	30
10.17	colorstretch	30
10.17.1	colorstretch – preliminaries	31
10.18	zebranize	33
10.18.1	zebranize – preliminaries	34
10.18.2	zebranize – the function	34
11	Drawing	36
12	Known Bugs	38
13	To Dos	38
14	Literature	38
15	Thanks	38

Part I

User Documentation

1 How It Works

We make use of Lua_T_E_Xs 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 $\text{T}_{\text{E}}\text{X}$ side or use the Lua functions directly. In fact, the $\text{T}_{\text{E}}\text{X}$ macros are simple wrappers around the functions.

2.1 $\text{T}_{\text{E}}\text{X}$ 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 `\chickenizesetup` 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 10th chicken is uppercase. However, the beginning of a sentence is not recognized automatically.²

`\dubstepize` wub wub wub wub wub BROOOOOAR WOBBBWOBWBWOB BZZZRRRRRRROOOOOOAAAAA ... (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.

²If 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 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 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 its 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.³

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 have⁴ a `\text-`version that takes an argument. `\textrandomcolor{foo}` results in a colored `foo` while the rest of the document remains unaffected.

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

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

However, to achieve this effect, still the whole 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.⁵

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,

⁵On a 500 pages text-only \LaTeX document the dilation is on the order of 10% with `\textrandomcolor`, but other manipulations can take much more time. However, you are not supposed to make such long documents with `chickenize`!

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. <<<<<<< HEAD

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`).

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

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

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

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 code points of the characters, e.g. `leetable[101] = 50` replaces every e (101) with the number 3 (50).

uclcratio = **<float>** 0.5 Gives the fraction of uppercases to lowercases in the `\randomuclc` mode. A higher number (up to 1) will result in more uppercase letters. Guess what a lower number does. >>>>>>> d9c88b6094abe878dd97cfd77f26e514b1d66c63

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_T_EX it's just to get an idea how things work here. For a deeper understanding of Lua_T_EX you should consult both the Lua_T_EX 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_T_EX 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 _T_EXing, especially the `tex.` library that offers access to _T_EX internals. In the simple example above, the function `tex.print()` inserts its argument into the _T_EX 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 _T_EX code, but rather in a separate file. It can then be loaded using

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

If you use Lua_A_T_EX, 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 _T_EX behaves: The *callbacks*. A callback is a point where you can hook into _T_EX'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 _T_EX'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.) _T_EX 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 _T_EX'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 address 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 address 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 LuaTeX 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 T_EXing or even for good LuaT_EXing. 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 T_EX 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 LuaT_EX's attributes.

For (un)registering, we use the `luatexbase` package. Then, the `.lua` file is loaded which does the actual work. Finally, the T_EX 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] = "BZZZRRRRRRRROOOOOOAAAAA"
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",gutenbergenize_rq,"gutenbergenize"}

```

```

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{luaexbase.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\randfontsassr
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\textrandomfontss#1%
185   {\setluatexattribute\randfontsassr{42}#1\unsetluatexattribute\randfontsassr}
186 \long\def\textrandomfontss#1%
187   {\setluatexattribute\randfontsassr{42}#1\unsetluatexattribute\randfontsassr}
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^AT_EX package

I have decided to keep the L^AT_EX-part of this package as small as possible. So far, it does ... nothing useful, but it provides a `chickenize.sty` that loads `chickenize.tex` 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.

```

235 \ProvidesPackage{chickenize}%
236 [2012/05/20 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 at the 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, otherwise we lose the freedom to change the string in-docum
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         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         elseif chickencount then
345             chicken_substitutions = chicken_substitutions + 1
346         end
347     end
348     return head
349 end
350
351 local separator    = string.rep("=", 28)
352 local texiowrite_nl = texio.write_nl
353 nicetext = function()
354     texiowrite_nl("Output written on "..tex.jobname.."pdf ("..status.total_pages.." chicken,".." e
355     texiowrite_nl(" ")
356     texiowrite_nl(separator)
357     texiowrite_nl("Hello my dear user,")
358     texiowrite_nl("good job, now go outside and enjoy the world!")
359     texiowrite_nl(" ")
360     texiowrite_nl("And don't forget to feed your chicken!")
361     texiowrite_nl(separator .. "\n")
362     if chickencount then
363         texiowrite_nl("There were "..chicken_substitutions.." substitutions made.")
364         texiowrite_nl(separator)
365     end

```

366 end

10.2 guttenbergenize

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

Calling `\gutenbergenize` 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.

```
367 local quotestrings = {
368   [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,
373 }
```

10.2.2 guttenbergenize – the function

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

10.3 hammertime

This is a completely useless function. It just prints STOP! – HAMMERTIME at the beginnig 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 $\text{Lua}\TeX$ mailing list.⁷

```
383 hammertimedelay = 1.2
384 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
```

⁶Thanks to Jasper for bringing me to this idea!

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

```

385 hammertime = function(head)
386   if hammerfirst then
387     texiowrite_nl(htime_separator)
388     texiowrite_nl("=====STOP!=====\\n")
389     texiowrite_nl(htime_separator .. "\\n\\n\\n")
390     os.sleep (hammertimedelay*1.5)
391     texiowrite_nl(htime_separator .. "\\n")
392     texiowrite_nl("=====HAMMERTIME=====\\n")
393     texiowrite_nl(htime_separator .. "\\n\\n")
394     os.sleep (hammertimedelay)
395     hammerfirst = false
396   else
397     os.sleep (hammertimedelay)
398     texiowrite_nl(htime_separator)
399     texiowrite_nl("=====U can't touch this!=====\\n")
400     texiowrite_nl(htime_separator .. "\\n\\n")
401     os.sleep (hammertimedelay*0.5)
402   end
403   return head
404 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.

```

405 itsame = function()
406 local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
407 color = "1 .6 0"
408 for i = 6,9 do mr(i,3) end
409 for i = 3,11 do mr(i,4) end
410 for i = 3,12 do mr(i,5) end
411 for i = 4,8 do mr(i,6) end
412 for i = 4,10 do mr(i,7) end
413 for i = 1,12 do mr(i,11) end
414 for i = 1,12 do mr(i,12) end
415 for i = 1,12 do mr(i,13) end
416
417 color = ".3 .5 .2"
418 for i = 3,5 do mr(i,3) end mr(8,3)
419 mr(2,4) mr(4,4) mr(8,4)
420 mr(2,5) mr(4,5) mr(5,5) mr(9,5)
421 mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
422 for i = 3,8 do mr(i,8) end
423 for i = 2,11 do mr(i,9) end
424 for i = 1,12 do mr(i,10) end
425 mr(3,11) mr(10,11)

```

```

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

```

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

```

10.6 leetspeak

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

```

455 leetspeak_onlytext = false
456 leettable = {
457   [101] = 51, -- E
458   [105] = 49, -- I

```



```

459 [108] = 49, -- L
460 [111] = 48, -- O
461 [115] = 53, -- S
462 [116] = 55, -- T
463
464 [101-32] = 51, -- e
465 [105-32] = 49, -- i
466 [108-32] = 49, -- l
467 [111-32] = 48, -- o
468 [115-32] = 53, -- s
469 [116-32] = 55, -- t
470 }

```

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

```

471 leet = function(head)
472   for line in nodetraverseid(Hhead,head) do
473     for i in nodetraverseid(GLYPH,line.head) do
474       if not leetspeak_onlytext or
475         node.has_attribute(i,luatexbase.attributes.leetattr)
476       then
477         if leettable[i.char] then
478           i.char = leettable[i.char]
479         end
480       end
481     end
482   end
483   return head
484 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

```

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

```

```
492 letterspace_pen.penalty = 10000
```

10.7.2 function implementation

```
493 letterspaceadjust = function(head)
494   for glyph in nodetraverseid(nodeid"glyph", head) do
495     if glyph.prev and (glyph.prev.id == nodeid"glyph" or glyph.prev.id == nodeid"disc") then
496       local g = nodecopy(letterspace_glue)
497       nodeinsertbefore(head, glyph, g)
498       nodeinsertbefore(head, g, nodecopy(letterspace_pen))
499     end
500   end
501   return head
502 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.

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

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

10.11 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

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

10.12 randomchars

```

556 randomchars = function(head)
557   for line in nodetraverseid(Hhead,head) do
558     for i in nodetraverseid(GLYPH,line.head) do
559       i.char = math.floor(math.random()*512)
560     end
561   end
562   return head
563 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.

```

564 randomcolor_grey = false
565 randomcolor_onlytext = false --switch between local and global colorization
566 rainbowcolor = false
567
568 grey_lower = 0
569 grey_upper = 900
570
571 Rgb_lower = 1
572 rGb_lower = 1
573 rgB_lower = 1
574 Rgb_upper = 254
575 rGb_upper = 254
576 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.

```

577 rainbow_step = 0.005
578 rainbow_Rgb = 1-rainbow_step -- we start in the red phase
579 rainbow_rGb = rainbow_step -- values x must always be  $0 < x < 1$ 
580 rainbow_rgB = rainbow_step
581 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.

```

582 randomcolorstring = function()
583   if randomcolor_grey then
584     return (0.001*math.random(grey_lower,grey_upper)).." g"
585   elseif rainbowcolor then
586     if rainind == 1 then -- red
587       rainbow_rGb = rainbow_rGb + rainbow_step
588       if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
589     elseif rainind == 2 then -- yellow
590       rainbow_Rgb = rainbow_Rgb - rainbow_step
591       if rainbow_Rgb <= rainbow_step then rainind = 3 end

```

```

592     elseif rainind == 3 then -- green
593         rainbow_rgB = rainbow_rgB + rainbow_step
594         rainbow_rGb = rainbow_rGb - rainbow_step
595         if rainbow_rGb <= rainbow_step then rainind = 4 end
596     elseif rainind == 4 then -- blue
597         rainbow_Rgb = rainbow_Rgb + rainbow_step
598         if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
599     else -- purple
600         rainbow_rgB = rainbow_rgB - rainbow_step
601         if rainbow_rgB <= rainbow_step then rainind = 1 end
602     end
603     return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rgB.." rg"
604 else
605     Rgb = math.random(Rgb_lower,Rgb_upper)/255
606     rGb = math.random(rGb_lower,rGb_upper)/255
607     rgB = math.random(rgB_lower,rgB_upper)/255
608     return Rgb.." "..rGb.." "..rgB.." .." rg"
609 end
610 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.

```

611 randomcolor = function(head)
612   for line in nodetraverseid(0,head) do
613     for i in nodetraverseid(37,line.head) do
614       if not(randomcolor_onlytext) or
615         (node.has_attribute(i,luatexbase.attributes.randcolorattr))
616       then
617         color_push.data = randomcolorstring() -- color or grey string
618         line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
619         nodeinsertafter(line.head,i,nodecopy(color_pop))
620       end
621     end
622   end
623   return head
624 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, next to nothing will be visible. Should be extended to also remove rules or just anything visible.

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

10.16 uppercasecolor

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

```
640 uppercasecolor = function (head)
641   for line in nodetraverseid(Hhead,head) do
642     for upper in nodetraverseid(GLYPH,line.head) do
643       if (((upper.char > 64) and (upper.char < 91)) or
644         ((upper.char > 57424) and (upper.char < 57451))) then -- for small caps! nice
645         color_push.data = randomcolorstring() -- color or grey string
646         line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
647         nodeinsertafter(line.head,upper,nodecopy(color_pop))
648       end
649     end
650   end
651   return head
652 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 \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!

10.17.1 `colorstretch` – preliminaries

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

```

653 keeptext = true
654 colorexpansion = true
655
656 colorstretch_coloroffset = 0.5
657 colorstretch_colorange = 0.5
658 chickenize_rule_bad_height = 4/5 -- height and depth of the rules
659 chickenize_rule_bad_depth = 1/5
660
661
662 colorstretchnumbers = true
663 drawstretchthreshold = 0.1
664 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 (`colorexpansion == true`), then the first glyph node is determined and its width compared with the width of the unexpanded glyph. This gives a measure for the expansion factor and is translated into a grey scale.

```

665 colorstretch = function (head)
666   local f = font.getfont(font.current()).characters
667   for line in nodetraverseid(Hhead,head) do
668     local rule_bad = nodenew(RULE)
669
670     if colorexpansion then -- if also the font expansion should be shown
671       local g = line.head
672       while not(g.id == 37) do
673         g = g.next
674       end
675       exp_factor = g.width / f[g.char].width
676       exp_color = colorstretch_coloroffset + (1-exp_factor)*10 .. " g"
677       rule_bad.width = 0.5*line.width -- we need two rules on each line!
678     else
679       rule_bad.width = line.width -- only the space expansion should be shown, only one rule
680     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.

```

681 rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bet
682 rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth
683
684 local glue_ratio = 0
685 if line.glue_order == 0 then
686     if line.glue_sign == 1 then
687         glue_ratio = colorstretch_colorrage * math.min(line.glue_set,1)
688     else
689         glue_ratio = -colorstretch_colorrage * math.min(line.glue_set,1)
690     end
691 end
692 color_push.data = colorstretch_coloroffset + glue_ratio .. " g"
693

```

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

```

694 -- set up output
695 local p = line.head
696
697 -- a rule to immitate kerning all the way back
698 local kern_back = nodenew(RULE)
699 kern_back.width = -line.width
700
701 -- if the text should still be displayed, the color and box nodes are inserted additionally
702 -- and the head is set to the color node
703 if keeptext then
704     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
705 else
706     node.flush_list(p)
707     line.head = nodecopy(color_push)
708 end
709 nodeinsertafter(line.head,line.head,rule_bad) -- then the rule
710 nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
711 tmpnode = nodeinsertafter(line.head,line.head.next.next,kern_back)
712
713 -- then a rule with the expansion color
714 if colorexansion then -- if also the stretch/shrink of letters should be shown
715     color_push.data = exp_color
716     nodeinsertafter(line.head,tmpnode,nodecopy(color_push))
717     nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
718     nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
719 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.

```

720 if colorstretchnumbers then

```



```

721     j = 1
722     glue_ratio_output = {}
723     for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
724         local char = unicode.utf8.char(s)
725         glue_ratio_output[j] = nodenew(37,1)
726         glue_ratio_output[j].font = font.current()
727         glue_ratio_output[j].char = s
728         j = j+1
729     end
730     if math.abs(glue_ratio) > drawstretchthreshold then
731         if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
732         else color_push.data = "0 0.99 0 rg" end
733     else color_push.data = "0 0 0 rg"
734     end
735
736     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
737     for i = 1,math.min(j-1,7) do
738         nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])
739     end
740     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
741 end -- end of stretch number insertion
742 end
743 return head
744 end

```

dubstepize

BROOOAR WOBBOWBWOBB BROOOOAR WOBBOWBWOBB BROOOOAR WOBB WOBB WOBB ...

745

scorpionize

These functions intentionally not documented.

```

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

```

752 zebracolorarray = {}
753 zebracolorarray_bg = {}
754 zebracolorarray[1] = "0.1 g"
755 zebracolorarray[2] = "0.9 g"
756 zebracolorarray_bg[1] = "0.9 g"
757 zebracolorarray_bg[2] = "0.1 g"

```

10.18.2 zebranize – the function

This code has to be revisited, it is ugly.

```

758 function zebranize(head)
759   zebracolor = 1
760   for line in nodetraverseid(nodeid"hhead",head) do
761     if zebracolor == #zebracolorarray then zebracolor = 0 end
762     zebracolor = zebracolor + 1
763     color_push.data = zebracolorarray[zebracolor]
764     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
765     for n in nodetraverseid(nodeid"glyph",line.head) do
766       if n.next then else
767         nodeinsertafter(line.head,n,nodecopy(color_pull))
768       end
769     end
770
771     local rule_zebra = nodenew(RULE)
772     rule_zebra.width = line.width
773     rule_zebra.height = tex.baselineskip.width*4/5
774     rule_zebra.depth = tex.baselineskip.width*1/5
775
776     local kern_back = nodenew(RULE)
777     kern_back.width = -line.width
778
779     color_push.data = zebracolorarray_bg[zebracolor]
780     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
781     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
782     nodeinsertafter(line.head,line.head,kern_back)
783     nodeinsertafter(line.head,line.head,rule_zebra)
784   end
785   return (head)
786 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!

```
787 --
788 function pdf_print (...)
789   for _, str in ipairs({...}) do
790     pdf.print(str .. " ")
791   end
792   pdf.print("\string\n")
793 end
794
795 function move (p)
796   pdf_print(p[1],p[2],"m")
797 end
798
799 function line (p)
800   pdf_print(p[1],p[2],"l")
801 end
802
803 function curve(p1,p2,p3)
804   pdf_print(p1[1], p1[2],
805             p2[1], p2[2],
806             p3[1], p3[2], "c")
807 end
808
809 function close ()
810   pdf_print("h")
811 end
812
813 function linewidth (w)
814   pdf_print(w,"w")
815 end
816
817 function stroke ()
818   pdf_print("S")
819 end
820 --
```

```

821
822 function strictcircle(center,radius)
823   local left = {center[1] - radius, center[2]}
824   local lefttop = {left[1], left[2] + 1.45*radius}
825   local leftbot = {left[1], left[2] - 1.45*radius}
826   local right = {center[1] + radius, center[2]}
827   local righttop = {right[1], right[2] + 1.45*radius}
828   local rightbot = {right[1], right[2] - 1.45*radius}
829
830   move (left)
831   curve (lefttop, righttop, right)
832   curve (rightbot, leftbot, left)
833 stroke()
834 end
835
836 function disturb_point(point)
837   return {point[1] + math.random()*5 - 2.5,
838           point[2] + math.random()*5 - 2.5}
839 end
840
841 function sloppycircle(center,radius)
842   local left = disturb_point({center[1] - radius, center[2]})
843   local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
844   local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}
845   local right = disturb_point({center[1] + radius, center[2]})
846   local righttop = disturb_point({right[1], right[2] + 1.45*radius})
847   local rightbot = disturb_point({right[1], right[2] - 1.45*radius})
848
849   local right_end = disturb_point(right)
850
851   move (right)
852   curve (rightbot, leftbot, left)
853   curve (lefttop, righttop, right_end)
854   linewidth(math.random()+0.5)
855   stroke()
856 end
857
858 function sloppyline(start,stop)
859   local start_line = disturb_point(start)
860   local stop_line = disturb_point(stop)
861   start = disturb_point(start)
862   stop = disturb_point(stop)
863   move(start) curve(start_line,stop_line,stop)
864   linewidth(math.random()+0.5)
865   stroke()
866 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:

- 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, 1st 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. I also think Philipp Gesang who gave me many advices on the Lua code – which I still didn't have time to correct ...