

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

Paul Isambert

CHICKENIZE

v0.1a

Arno Trautmann

arno.trautmann@gmx.de

This is the documentation of the package `chickenize`. It allows manipulations of any $\text{Lua}\TeX$ document¹ exploiting the possibilities offered by the callbacks that influence line breaking (and some other stuff). 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 shortly informs you about some of your possibilities and provides links to the (documented) Lua functions. The \TeX interface is presented [below](#).

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

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

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

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

This package is copyright © 2012 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 'maintained'.

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

For the Impatient:

A small and incomplete overview of the functionalities offered by this package. I try to keep this list as complete as possible.² Of course, the label “complete nonsense” depends on what you are doing ...

maybe useful functions

<code>colorstretch</code>	shows grey boxes that visualise the badness and font expansion of each line
<code>letterspaceadjust</code>	improves the greyness by using a small amount of letterspacing
<code>substitutewords</code>	replaces words by other words (user-controlled!)

less useful functions

<code>boustrophedon</code>	invert every second line in the style of archaic greek texts
<code>countglyphs</code>	counts the number of glyphs in the whole document
<code>leetspeak</code>	translates the (latin-based) input into 1337 5p34k
<code>randomucl</code>	alternates randomly between uppercase and lowercase
<code>rainbowcolor</code>	changes the color of letters slowly according to a rainbow
<code>randomcolor</code>	prints every letter in a random color
<code>tabularasa</code>	removes every glyph from the output and leaves an empty document
<code>uppercasecolor</code>	makes every uppercase letter colored

complete nonsense

<code>chickenize</code>	replaces every word with “chicken” (or user-adjustable words)
<code>gutenbergize</code>	deletes every quote and footnotes
<code>hammertime</code>	U can't touch this!
<code>kernmanipulate</code>	manipulates the kerning (tbi)
<code>matrixize</code>	replaces every glyph by its ASCII value in binary code
<code>randomerror</code>	just throws random (La)TeX errors at random times
<code>randomfonts</code>	changes the font randomly between every letter
<code>randomchars</code>	randomizes the (letters of the) whole input

²If you notice that something is missing, please help me improving the documentation!

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	7
2.3	\text-Versions	7
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	TeX package	20
9.1	Definition of User-Level Macros	20
10	Lua Module	21
10.1	chickenize	21
10.2	boustrophedon	24
10.3	countglyphs	25
10.4	gutenbergize	26
10.4.1	gutenbergize – preliminaries	26
10.4.2	gutenbergize – the function	26
10.5	hammertime	26
10.6	itsame	27
10.7	kernmanipulate	28
10.8	leetspeak	29
10.9	letterspaceadjust	29
10.9.1	setup of variables	30

10.9.2	function implementation	30
10.9.3	textletterspaceadjust	30
10.10	matrixize	30
10.11	pancakenize	31
10.12	randomerror	31
10.13	randomfonts	31
10.14	randomuclc	32
10.15	randomchars	32
10.16	randomcolor and rainbowcolor	33
10.16.1	randomcolor – preliminaries	33
10.16.2	randomcolor – the function	34
10.17	randomerror	34
10.18	rickroll	34
10.19	substitutewords	34
10.20	tabularasa	35
10.21	uppercasecolor	35
10.22	colorstretch	36
10.22.1	colorstretch – preliminaries	36
10.23	zebranize	39
10.23.1	zebranize – preliminaries	39
10.23.2	zebranize – the function	39
11	Drawing	41
12	Known Bugs	43
13	To Do's	43
14	Literature	43
15	Thanks	43

Part I

User Documentation

1 How It Works

We make use of Lua \TeX s callbacks, especially the `pre_linebreak_filter` and the `post_linebreak_filter`. Hooking a function into these, we can nearly arbitrarily change the content of the document. If the changes should be on the input-side (e. g. replacing words with `chicken`), one can use the `pre_linebreak_filter`. However, changes like inserting color are best made after the linebreak is finalized, so `post_linebreak_filter` is to be preferred for such things.

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

2 Commands – How You Can Use It

There are several ways to make use of the *chickenize* package – you can either stay on the \TeX side or use the Lua functions directly. In fact, the \TeX macros are simple wrappers around the functions.

2.1 \TeX Commands – Document Wide

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

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

`\boustrophedon` Reverts every second line. This immitates archaic greek writings where one line was right-to-left, the next one left-to-right etc.³ Interestingly, also every glyph was adaptet to the writing direction, so all glyphs are inverted in the right-to-left lines. Actually, there are two versions of this command that differ in their implementation: `\boustrophedon` rotates the whole line, while `\boustrophedonglyphs` changes the writing direction and reverses glyph-wise. The second one takes much more compilation time, but may be more reliable. A Rongorongo⁴ similar style boustrophedon is available with `\boustrophedoninverse` or `\rongorongonize`, where subsequent lines are rotated by 180° instead of mirrored.

`\countglyphs` Counts every printed character that appeared in anything that is a paragraph. Which is quite everything, in fact, *exept* math mode! The total number will be printed at the end of the log file/console output.

`\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.⁵

³en.wikipedia.org/wiki/Boustrophedon

⁴en.wikipedia.org/wiki/Rongorongo

⁵If you have a nice implementation idea, I'd love to include this!

\dubstepize wub wub wub wub wub BROOOOOAR WOBBBWOB BWOB 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.

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

\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.

\pancakenize This is a dummy command that does nothing. However, every time you use it, you owe a pancake to the package author. You can either send it via mail or bring it to some (local) \TeX user's group meeting.

\tabularasa Takes every glyph out of the document and replaces it by empty space of the same width. That could be useful if you want to hide some part of a text or similar. The \text-version is most likely more useful.

\leetspeak Translates the input into 1337 speak. If you don't understand that, lern it, n00b.

\nyanize A synonym for rainbowcolor.

\matrixize Replaces every glyph by a binary representation of its ASCII value.

\colorstretch Inspired by Paul Isambert's code, this command prints boxes instead of lines. The greyness of the first (left-hand) box corresponds to the badness of the line, i. e. it is a measure for how much the space between words has been extended to get proper paragraph justification. The second box on the right-hand side shows the amount of stretching/shrinking when font expansion is used. Together, the greyness of both boxes indicate how well the greyness is distributed over the typeset page.

\substitutewords You have to specify pairs of words by using \addtosubstitutions{word1}{word2}. Then call \substitutewords (or the other way round, doesn't matter) and each occurrence of word1 will be replaced by word2. You can add replacement pairs by repeated calls to \addtosubstitutions. Take care! This function works with the input directly, therefore it does *not* work on text that is inserted by macros, but it *will* work on macro names itself! This way, you may use it to change macros (or environments) at will. Bug or feature? I'm not sure right now ...

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.⁶

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 have⁷ a `\text-`version that takes an argument. `\textrandomcolor{foo}` results in a colored `foo` while the rest of the document remains unaffected. However, to achieve this effect, still the whole 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.⁸

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?

⁶Which is so far not catchable due to missing functionality in `luatexbase`.

⁷If they don't have, I did miss that, sorry. Please inform me about such cases.

⁸On 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`!

However, `\chickenizesetup` is a macro on the T_EX side meaning that you can use *only* % as comment string. If you use `--`, all of the argument will be ignored as T_EX does not pass an eol to `\directlua`. If you don't understand that, just ignore it and go on as usual.

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

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

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

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

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

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

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

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

`leetttable = <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. `leetttable[101] = 50` replaces every e (101) with the number 3 (50).

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

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

`rainbow_step = <float> 0.005` This indicates the relative change of color using the rainbow functionality. A value of 1 changes the color in one step from red to yellow, while a value of `0.005` takes 200 letters for the transition to be completed. Useful values are below `0.05`, but it depends on the amount of text. The longer the text and the lower the `step`, the nicer your rainbow will be.

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

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

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

Part II

Tutorial

I thought it might be helpful to add a small tutorial to this package as it is mainly written with instructional purposes in mind. However, the following is *not* intended as a comprehensive guide to Lua \TeX ! It's just to get an idea how things work here. For a deeper understanding of Lua \TeX you should consult both the Lua \TeX 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 \TeX 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 \TeX ing, especially the `tex.` library that offers access to \TeX internals. 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 embedded in your \TeX code, but rather in a separate file. It can then be loaded using

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

If you use Lua \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 to it that may make sense – or not. (Thus maybe breaking your document completely ...)

Callbacks are employed at several stages 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.) \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 \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 why we avoid this syntax here. Instead, we rely on the package `luatexbase` by Manuel Pégourié-Gonnard and Élie Roux that offers the function `luatexbase.add_to_callback` which has a somewhat extended syntax:

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

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

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

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

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

6 Nodes

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

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

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

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

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

```
function remove_e(head)
```

```

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

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

```

Now, don't read on, but try out this code by yourself! Change the number of the character to be removed, try to play around a bit. Also, try to remove the spaces between words. Those are glue nodes – look up their id in the 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 TeXing or even for good LuaTeXing. 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 \TeX file

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

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

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

```
1 \input{luatexbase.sty}
2 \directlua{dofile(kpse.find_file("chickenize.lua"))}
3
4 \def\BEclerize{
5   \chickenize
6   \directlua{
7     chickenstring[1] = "noise noise"
8     chickenstring[2] = "atom noise"
9     chickenstring[3] = "shot noise"
10    chickenstring[4] = "photon noise"
11    chickenstring[5] = "camera noise"
12    chickenstring[6] = "noising noise"
13    chickenstring[7] = "thermal noise"
14    chickenstring[8] = "electronic noise"
15    chickenstring[9] = "spin noise"
16    chickenstring[10] = "electron noise"
17    chickenstring[11] = "Bogoliubov noise"
18    chickenstring[12] = "white noise"
19    chickenstring[13] = "brown noise"
20    chickenstring[14] = "pink noise"
21    chickenstring[15] = "bloch sphere"
22    chickenstring[16] = "atom shot noise"
23    chickenstring[17] = "nature physics"
24   }
25 }
26
27 \def\boustrophedon{
28   \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon,"boustrophedon")}}
29 \def\unboustrophedon{
30   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon")}}
31
```

```

32 \def\boustrophendonglyphs{
33   \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon_glyphs,"boustrophedon_glyphs")}
34 \def\unboustrophendonglyphs{
35   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon_glyphs")}}
36
37 \def\boustrophedoninverse{
38   \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon_inverse,"boustrophedon_inverse")}
39 \def\unboustrophedoninverse{
40   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon_inverse")}}
41
42 \def\chickenize{
43   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize")}
44   luatexbase.add_to_callback("start_page_number",
45     function() texio.write("[..status.total_pages) end ,"cstartpage")
46     luatexbase.add_to_callback("stop_page_number",
47     function() texio.write(" chickens]") end,"cstoppage")
48 %
49   luatexbase.add_to_callback("stop_run",nicetext,"a nice text")
50 }
51 }
52 \def\unchickenize{
53   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","chickenize")}
54   luatexbase.remove_from_callback("start_page_number","cstartpage")
55   luatexbase.remove_from_callback("stop_page_number","cstoppage")}}
56
57 \def\coffeestainize{ %% to be implemented.
58   \directlua{}}
59 \def\uncoffeestainize{
60   \directlua{}}
61
62 \def\colorstretch{
63   \directlua{luatexbase.add_to_callback("post_linebreak_filter",colorstretch,"stretch_expansion")}
64 \def\uncolorstretch{
65   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","stretch_expansion")}}
66
67 \def\countglyphs{
68   \directlua{glyphnumber = 0
69     luatexbase.add_to_callback("post_linebreak_filter",countglyphs,"countglyphs")
70     luatexbase.add_to_callback("stop_run",printglyphnumber,"printglyphnumber")
71   }
72 }
73
74 \def\dosomethingfunny{
75   %% should execute one of the "funny" commands, but randomly. So every compilation is complete.
76 }
77

```

```

78 \def\dubstepenize{
79   \chickenize
80   \directlua{
81     chickenstring[1] = "WOB"
82     chickenstring[2] = "WOB"
83     chickenstring[3] = "WOB"
84     chickenstring[4] = "BROOOAR"
85     chickenstring[5] = "WHEE"
86     chickenstring[6] = "WOB WOB WOB"
87     chickenstring[7] = "WAAAAAAAAAH"
88     chickenstring[8] = "duhduh duhduh duh"
89     chickenstring[9] = "BEEEEEEEEEW"
90     chickenstring[10] = "DEEEEEEEEEW"
91     chickenstring[11] = "EEEEEW"
92     chickenstring[12] = "boop"
93     chickenstring[13] = "buhdee"
94     chickenstring[14] = "bee bee"
95     chickenstring[15] = "BZZRRRRRRRROOOOOOAAAAA"
96
97     chickenizefraction = 1
98   }
99 }
100 \let\dubstepize\dubstepenize
101
102 \def\gutenbergize{ %% makes only sense when using LaTeX
103   \AtBeginDocument{
104     \let\grqq\relax\let\glqq\relax
105     \let\frqq\relax\let\flqq\relax
106     \let\grq\relax\let\glq\relax
107     \let\frq\relax\let\flq\relax
108 %
109     \gdef\footnote##1{}
110     \gdef\cite##1{}\gdef\parencite##1{}
111     \gdef\Cite##1{}\gdef\Parencite##1{}
112     \gdef\cites##1{}\gdef\parencites##1{}
113     \gdef\Cites##1{}\gdef\Parencites##1{}
114     \gdef\footcite##1{}\gdef\footcitetext##1{}
115     \gdef\footcites##1{}\gdef\footcitetexts##1{}
116     \gdef\textcite##1{}\gdef\Textcite##1{}
117     \gdef\textcites##1{}\gdef\Textcites##1{}
118     \gdef\smartcites##1{}\gdef\Smartcites##1{}
119     \gdef\supercite##1{}\gdef\supercites##1{}
120     \gdef\autocite##1{}\gdef\Autocite##1{}
121     \gdef\autocites##1{}\gdef\Autocites##1{}
122     %% many, many missing ... maybe we need to tackle the underlying mechanism?
123   }

```

```

124 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",guttenbergenize_rq,"guttenbergenize_rq")}
125 }
126
127 \def\hammertime{
128   \global\let\n\relax
129   \directlua{hammerfirst = true
130             luatexbase.add_to_callback("pre_linebreak_filter",hammertime,"hammertime")}}
131 \def\unhammertime{
132   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","hammertime")}}
133
134 % \def\itsame{
135 %   \directlua{drawmario}} %% does not exist
136
137 \def\kernmanipulate{
138   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",kernmanipulate,"kernmanipulate")}}
139 \def\unkernmanipulate{
140   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter",kernmanipulate)}}
141
142 \def\leetspeak{
143   \directlua{luatexbase.add_to_callback("post_linebreak_filter",leet,"1337")}}
144 \def\unleetspeak{
145   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","1337")}}
146
147 \def\letterspaceadjust{
148   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",letterspaceadjust,"letterspaceadjust")}}
149 \def\unletterspaceadjust{
150   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter",letterspaceadjust)}}
151
152 \def\listallcommands{
153   \directlua{
154     for name in pairs(tex.hashtokens()) do
155       print(name)
156     end}
157 }
158
159 \let\stealsheep\letterspaceadjust %% synonym in honor of Paul
160 \let\unstealsheep\unletterspaceadjust
161 \let\returnsheep\unletterspaceadjust
162
163 \def\matrixize{
164   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",matrixize,"matrixize")}}
165 \def\unmatrixize{
166   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter",matrixize)}}
167
168 \def\milkcow{ %% FIXME %% to be implemented
169   \directlua{}}

```



```

170 \def\unmilkcow{
171   \directlua{}}
172
173 \def\pancakelize{
174   \directlua{luatexbase.add_to_callback("stop_run",pancaketext,"pancaketext")}}
175
176 \def\rainbowcolor{
177   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"rainbowcolor")
178     rainbowcolor = true}}
179 \def\unrainbowcolor{
180   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","rainbowcolor")
181     rainbowcolor = false}}
182 \let\nyanize\rainbowcolor
183 \let\unnyanize\unrainbowcolor
184
185 \def\randomcolor{
186   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"randomcolor")}}
187 \def\unrandomcolor{
188   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomcolor")}}
189
190 \def\randomerror{ %% FIXME
191   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomerror,"randomerror")}}
192 \def\unrandomerror{ %% FIXME
193   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomerror")}}
194
195 \def\randomfonts{
196   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomfonts,"randomfonts")}}
197 \def\unrandomfonts{
198   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomfonts")}}
199
200 \def\randomuclc{
201   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",randomuclc,"randomuclc")}}
202 \def\unrandomuclc{
203   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","randomuclc")}}
204
205 \let\rongorongonize\boustrophedoninverse
206 \let\unrongorongonize\unboustrophedoninverse
207
208 \def\scorpionize{
209   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",scorpionize_color,"scorpionize_color")}}
210 \def\unscorpionize{
211   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","scorpionize_color")}}
212
213 \def\spankmonkey{ %% to be implemented
214   \directlua{}}
215 \def\unspankmonkey{

```

```

216 \directlua{}}
217
218 \def\substitutewords{
219   \directlua{luatexbase.add_to_callback("process_input_buffer",substitutewords,"substitutewords")}
220 \def\unsubstitutewords{
221   \directlua{luatexbase.remove_from_callback("process_input_buffer","substitutewords")}}
222
223 \def\addtosubstitutions#1#2{
224   \directlua{addtosubstitutions("#1","#2")}
225 }
226
227 \def\tabularasa{
228   \directlua{luatexbase.add_to_callback("post_linebreak_filter",tabularasa,"tabularasa")}}
229 \def\untabularasa{
230   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","tabularasa")}}
231
232 \def\uppercasecolor{
233   \directlua{luatexbase.add_to_callback("post_linebreak_filter",uppercasecolor,"uppercasecolor")}}
234 \def\unuppercasecolor{
235   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","uppercasecolor")}}
236
237 \def\zebranize{
238   \directlua{luatexbase.add_to_callback("post_linebreak_filter",zebranize,"zebranize")}}
239 \def\unzebranize{
240   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","zebranize")}}

```

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

```

241 \newluatexattribute\leetattr
242 \newluatexattribute\letterspaceadjustattr
243 \newluatexattribute\randcolorattr
244 \newluatexattribute\randfontssattr
245 \newluatexattribute\randuclcatr
246 \newluatexattribute\tabularasaattr
247 \newluatexattribute\uppercasecolorattr
248
249 \long\def\textleetspeak#1%
250   {\setluatexattribute\leetattr{42}#1\unsetluatexattribute\leetattr}
251
252 \long\def\textletterspaceadjust#1{
253   \setluatexattribute\letterspaceadjustattr{42}#1\unsetluatexattribute\letterspaceadjustattr
254   \directlua{
255     if (textletterspaceadjustactive) then else % -- if already active, do nothing
256       luatexbase.add_to_callback("pre_linebreak_filter",textletterspaceadjust,"textletterspaceadjust")
257     end
258     textletterspaceadjustactive = true           % -- set to active
259   }

```

```

260 }
261 \let\textlsa\textletterspaceadjust
262
263 \long\def\textrandomcolor#1%
264   {\setluatexattribute\randcolorattr{42}#1\unsetluatexattribute\randcolorattr}
265 \long\def\textrandomfonts#1%
266   {\setluatexattribute\randfontsattr{42}#1\unsetluatexattribute\randfontsattr}
267 \long\def\textrandomfonts#1%
268   {\setluatexattribute\randfontsattr{42}#1\unsetluatexattribute\randfontsattr}
269 \long\def\textrandomuclc#1%
270   {\setluatexattribute\randuclcatr{42}#1\unsetluatexattribute\randuclcatr}
271 \long\def\texttabularasa#1%
272   {\setluatexattribute\tabularasaattr{42}#1\unsetluatexattribute\tabularasaattr}
273 \long\def\textuppercasecolor#1%
274   {\setluatexattribute\uppercasecolorattr{42}#1\unsetluatexattribute\uppercasecolorattr}

```

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

```

275 \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.

```

276 \long\def\luadraw#1#2{%
277   \vbox to #1bp{%
278     \vfil
279     \luatexlualua{pdf_print("q") #2 pdf_print("Q")}%
280   }%
281 }
282 \long\def\drawchicken{
283   \luadraw{90}{
284     kopf = {200,50} % Kopfmitte
285     kopf_rad = 20
286
287     d = {215,35} % Halsansatz
288     e = {230,10} %
289
290     korper = {260,-10}
291     korper_rad = 40
292
293     bein11 = {260,-50}
294     bein12 = {250,-70}
295     bein13 = {235,-70}
296
297     bein21 = {270,-50}
298     bein22 = {260,-75}
299     bein23 = {245,-75}
300

```

```

301 schnabel_oben = {185,55}
302 schnabel_vorne = {165,45}
303 schnabel_unten = {185,35}
304
305 flugel_vorne = {260,-10}
306 flugel_unten = {280,-40}
307 flugel_hinten = {275,-15}
308
309 sloppycircle(kopf,kopf_rad)
310 sloppyline(d,e)
311 sloppycircle(korper,korper_rad)
312 sloppyline(bein11,bein12) sloppyline(bein12,bein13)
313 sloppyline(bein21,bein22) sloppyline(bein22,bein23)
314 sloppyline(schnabel_vorne,schnabel_oben) sloppyline(schnabel_vorne,schnabel_unten)
315 sloppyline(flugel_vorne,flugel_unten) sloppyline(flugel_hinten,flugel_unten)
316 }
317 }

```

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.

```

318 \ProvidesPackage{chickenize}%
319 [2012/09/16 v0.1a chickenize package]
320 \input{chickenize}

```

9.1 Definition of User-Level Macros

Nothing done so far, just some minor ideas. If you want to implement some cool things, contact me! :)

```

321 \iffalse
322 \DeclareDocumentCommand\includegraphics{0}{m}{
323   \fbox{Chicken} %% actually, I'd love to draw an MP graph showing a chicken ...
324 }
325 %%% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
326 %% So far, you have to load pgfplots yourself.
327 %% As it is a mighty package, I don't want the user to force loading it.
328 \NewDocumentCommand\balmerpeak{G{}0{-4cm}}{
329 %% to be done using Lua drawing.
330 }
331 \fi

```

10 Lua Module

This file contains all the necessary functions and is the actual work horse of this package. The functions are sorted strictly alphabetically (or, they *should* be ...) and not by sense, functionality or anything.

First, we set up some constants that are used by many of the following functions. These are made global so the code can be manipulated at the document level, too.

```
332
333 local nodenew = node.new
334 local nodecopy = node.copy
335 local nodeinsertbefore = node.insert_before
336 local nodeinsertafter = node.insert_after
337 local noderemove = node.remove
338 local nodeid = node.id
339 local nodetraverseid = node.traverse_id
340 local nodeslide = node.slide
341
342 Hhead = nodeid("hhead")
343 RULE = nodeid("rule")
344 GLUE = nodeid("glue")
345 WHAT = nodeid("whatsit")
346 COL = node.subtype("pdf_colorstack")
347 GLYPH = nodeid("glyph")
```

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

```
348 color_push = nodenew(WHAT,COL)
349 color_pop = nodenew(WHAT,COL)
350 color_push.stack = 0
351 color_pop.stack = 0
352 color_push.cmd = 1
353 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.

```
354 chicken_pagenumbers = true
355
356 chickenstring = {}
357 chickenstring[1] = "chicken" -- chickenstring is a table, please remeber this!
358
359 chickenizefraction = 0.5
360 -- set this to a small value to fool somebody, or to see if your text has been read carefully. Th
361 chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing you
362
363 local tbl = font.getfont(font.current())
```

```

364 local space = tbl.parameters.space
365 local shrink = tbl.parameters.space_shrink
366 local stretch = tbl.parameters.space_stretch
367 local match = unicode.utf8.match
368 chickenize_ignore_word = false

The function chickenize_real_stuff is started once the beginning of a to-be-substituted word is found.
369 chickenize_real_stuff = function(i,head)
370     while ((i.next.id == 37) or (i.next.id == 11) or (i.next.id == 7) or (i.next.id == 0)) do --
371         i.next = i.next.next
372     end
373
374     chicken = {} -- constructing the node list.
375
376 -- Should this be done only once? No, otherwise we lose the freedom to change the string in-docum
377 -- But it could be done only once each paragraph as in-paragraph changes are not possible!
378
379     chickenstring_tmp = chickenstring[math.random(1,#chickenstring)]
380     chicken[0] = nodenew(37,1) -- only a dummy for the loop
381     for i = 1,string.len(chickenstring_tmp) do
382         chicken[i] = nodenew(37,1)
383         chicken[i].font = font.current()
384         chicken[i-1].next = chicken[i]
385     end
386
387     j = 1
388     for s in string.utfvalues(chickenstring_tmp) do
389         local char = unicode.utf8.char(s)
390         chicken[j].char = s
391         if match(char,"%s") then
392             chicken[j] = nodenew(10)
393             chicken[j].spec = nodenew(47)
394             chicken[j].spec.width = space
395             chicken[j].spec.shrink = shrink
396             chicken[j].spec.stretch = stretch
397         end
398         j = j+1
399     end
400
401     nodeslide(chicken[1])
402     lang.hyphenate(chicken[1])
403     chicken[1] = node.kerning(chicken[1]) -- FIXME: does not work
404     chicken[1] = node.ligaturing(chicken[1]) -- dito
405
406     nodeinsertbefore(head,i,chicken[1])
407     chicken[1].next = chicken[2] -- seems to be necessary ... to be fixed
408     chicken[string.len(chickenstring_tmp)].next = i.next

```

```

409
410     -- shift lowercase latin letter to uppercase if the original input was an uppercase
411     if (chickenize_capital and (chicken[1].char > 96 and chicken[1].char < 123)) then
412         chicken[1].char = chicken[1].char - 32
413     end
414
415     return head
416 end
417
418 chickenize = function(head)
419     for i in nodetraverseid(37,head) do --find start of a word
420         if (chickenize_ignore_word == false) then -- normal case: at the beginning of a word, we jump
421             if (i.char > 64 and i.char < 91) then chickenize_capital = true else chickenize_capital = false
422             head = chickenize_real_stuff(i,head)
423         end
424
425         -- At the end of the word, the ignoring is reset. New chance for everyone.
426         if not((i.next.id == 37) or (i.next.id == 7) or (i.next.id == 22) or (i.next.id == 11)) then
427             chickenize_ignore_word = false
428         end
429
430         -- And the random determination of the chickenization of the next word:
431         if math.random() > chickenizefraction then
432             chickenize_ignore_word = true
433         elseif chickencount then
434             chicken_substitutions = chicken_substitutions + 1
435         end
436     end
437     return head
438 end
439

```

A small additional feature: Some nice text to cheer up the user. Mainly to show that and how we can access the `stop_run` callback. (see above)

```

440 local separator      = string.rep("=", 28)
441 local texiowrite_nl = texio.write_nl
442 nicetext = function()
443     texiowrite_nl("Output written on "..tex.jobname.."pdf ("..status.total_pages.." chicken,".." eg
444     texiowrite_nl(" ")
445     texiowrite_nl(separator)
446     texiowrite_nl("Hello my dear user,")
447     texiowrite_nl("good job, now go outside and enjoy the world!")
448     texiowrite_nl(" ")
449     texiowrite_nl("And don't forget to feed your chicken!")
450     texiowrite_nl(separator .. "\n")
451     if chickencount then
452         texiowrite_nl("There were "..chicken_substitutions.." substitutions made.")

```

```

453     texiowrite_nl(separator)
454 end
455 end

```

10.2 boustrophedon

There are two implementations of the boustrophedon: One reverses every line as a whole, the other one changes the writing direction and reverses glyphs one by one. The latter one might be more reliable, but takes considerably more time.

Linewise rotation:

```

456 boustrophedon = function(head)
457   rot = node.new(8,8)
458   rot2 = node.new(8,8)
459   odd = true
460   for line in node.traverse_id(0,head) do
461     if odd == false then
462       w = line.width/65536*0.99625 -- empirical correction factor (?)
463       rot.data = "-1 0 0 1 " .. w .. " 0 cm"
464       rot2.data = "-1 0 0 1 " .. -w .. " 0 cm"
465       line.head = node.insert_before(line.head,line.head,node.copy(rot))
466       node.insert_after(line.head,node.tail(line.head),node.copy(rot2))
467       odd = true
468     else
469       odd = false
470     end
471   end
472   return head
473 end

```

Glyphwise rotation:

```

474 boustrophedon_glyphs = function(head)
475   odd = false
476   rot = nodenew(8,8)
477   rot2 = nodenew(8,8)
478   for line in nodetraverseid(0,head) do
479     if odd==true then
480       line.dir = "TRT"
481       for g in nodetraverseid(37,line.head) do
482         w = -g.width/65536*0.99625
483         rot.data = "-1 0 0 1 " .. w .. " 0 cm"
484         rot2.data = "-1 0 0 1 " .. -w .. " 0 cm"
485         line.head = node.insert_before(line.head,g,node.copy(rot))
486         node.insert_after(line.head,g,node.copy(rot2))
487       end
488       odd = false
489     else
490       line.dir = "TLT"

```



```

491         odd = true
492     end
493 end
494 return head
495 end

```

Inverse boustrophedon. At least I think, this is the way Rongorongo is written. However, the top-to-bottom direction has to be inverted, too.

```

496 boustrophedon_inverse = function(head)
497     rot = node.new(8,8)
498     rot2 = node.new(8,8)
499     odd = true
500     for line in node.traverse_id(0,head) do
501         if odd == false then
502 texio.write_nl(line.height)
503             w = line.width/65536*0.99625 -- empirical correction factor (?)
504             h = line.height/65536*0.99625
505             rot.data = "-1 0 0 -1 "..w.." "..h.." cm"
506             rot2.data = "-1 0 0 -1"..-w.." "..0.5*h.." cm"
507             line.head = node.insert_before(line.head,line.head,node.copy(rot))
508             node.insert_after(line.head,node.tail(line.head),node.copy(rot2))
509             odd = true
510         else
511             odd = false
512         end
513     end
514 return head
515 end

```

10.3 countglyphs

Counts the glyphs in your documnt. Where “glyph” means every printed character in everything that is a paragraph – formulas do *not* work! However, hyphenations *do* work and the hyphen sign *is counted*! And that is the sole reason for this function – every simple script could read the letters in a doucment, but only after the hyphenation it is possible to count the real number of printed characters – where the hyphen does count.

This function will be extended to allow counting of whatever you want.

```

516 countglyphs = function(head)
517     for line in nodetraverseid(0,head) do
518         for glyph in nodetraverseid(37,line.head) do
519             glyphnumber = glyphnumber + 1
520         end
521     end
522 return head
523 end

```

To print out the number at the end of the document, the following function is registered in the stop_run

callback. This will prevent the normal message from being printed, informing the user about page and memory stats etc. But I guess when counting characters, everything else does not matter at all? ...

```
524 printglyphnumber = function()
525   texiowrite_nl("Number of glyphs in this document: "..glyphnumber.."\\n")
526 end
```

10.4 guttenbergenize

A function in honor of the German politician Gutenberg.⁹ 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.

10.4.1 guttenbergenize – preliminaries

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

```
527 local quotestrings = {
528   [171] = true, [172] = true,
529   [8216] = true, [8217] = true, [8218] = true,
530   [8219] = true, [8220] = true, [8221] = true,
531   [8222] = true, [8223] = true,
532   [8248] = true, [8249] = true, [8250] = true,
533 }
```

10.4.2 guttenbergenize – the function

```
534 guttenbergenize_rq = function(head)
535   for n in nodetraverseid(nodeid"glyph",head) do
536     local i = n.char
537     if quotestrings[i] then
538       noderemove(head,n)
539     end
540   end
541   return head
542 end
```

10.5 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

⁹Thanks to Jasper for bringing me to this idea!

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.¹⁰

```

543 hammertimedelay = 1.2
544 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
545 hammertime = function(head)
546   if hammerfirst then
547     texiowrite_nl(htime_separator)
548     texiowrite_nl("=====STOP!=====\\n")
549     texiowrite_nl(htime_separator .. "\\n\\n\\n")
550     os.sleep (hammertimedelay*1.5)
551     texiowrite_nl(htime_separator .. "\\n")
552     texiowrite_nl("=====HAMMERTIME=====\\n")
553     texiowrite_nl(htime_separator .. "\\n\\n")
554     os.sleep (hammertimedelay)
555     hammerfirst = false
556   else
557     os.sleep (hammertimedelay)
558     texiowrite_nl(htime_separator)
559     texiowrite_nl("=====U can't touch this!=====\\n")
560     texiowrite_nl(htime_separator .. "\\n\\n")
561     os.sleep (hammertimedelay*0.5)
562   end
563   return head
564 end

```

10.6 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.

```

565 itsame = function()
566 local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
567 color = "1 .6 0"
568 for i = 6,9 do mr(i,3) end
569 for i = 3,11 do mr(i,4) end
570 for i = 3,12 do mr(i,5) end
571 for i = 4,8 do mr(i,6) end
572 for i = 4,10 do mr(i,7) end
573 for i = 1,12 do mr(i,11) end
574 for i = 1,12 do mr(i,12) end
575 for i = 1,12 do mr(i,13) end
576
577 color = ".3 .5 .2"
578 for i = 3,5 do mr(i,3) end mr(8,3)
579 mr(2,4) mr(4,4) mr(8,4)

```

¹⁰<http://tug.org/pipermail/luatex/2011-November/003355.html>

```

580 mr(2,5) mr(4,5) mr(5,5) mr(9,5)
581 mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
582 for i = 3,8 do mr(i,8) end
583 for i = 2,11 do mr(i,9) end
584 for i = 1,12 do mr(i,10) end
585 mr(3,11) mr(10,11)
586 for i = 2,4 do mr(i,15) end for i = 9,11 do mr(i,15) end
587 for i = 1,4 do mr(i,16) end for i = 9,12 do mr(i,16) end
588
589 color = "1 0 0"
590 for i = 4,9 do mr(i,1) end
591 for i = 3,12 do mr(i,2) end
592 for i = 8,10 do mr(5,i) end
593 for i = 5,8 do mr(i,10) end
594 mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
595 for i = 4,9 do mr(i,12) end
596 for i = 3,10 do mr(i,13) end
597 for i = 3,5 do mr(i,14) end
598 for i = 7,10 do mr(i,14) end
599 end

```

10.7 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.

```

600 chickenkernamount = 0
601 chickeninvertkerning = false
602
603 function kernmanipulate (head)
604   if chickeninvertkerning then -- invert the kerning
605     for n in nodetraverseid(11,head) do
606       n.kern = -n.kern
607     end
608   else -- if not, set it to the given value
609     for n in nodetraverseid(11,head) do
610       n.kern = chickenkernamount
611     end
612   end
613   return head
614 end

```

10.8 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.

```
615 leetspeak_onlytext = false
616 leettable = {
617   [101] = 51, -- E
618   [105] = 49, -- I
619   [108] = 49, -- L
620   [111] = 48, -- O
621   [115] = 53, -- S
622   [116] = 55, -- T
623
624   [101-32] = 51, -- e
625   [105-32] = 49, -- i
626   [108-32] = 49, -- l
627   [111-32] = 48, -- o
628   [115-32] = 53, -- s
629   [116-32] = 55, -- t
630 }
```

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

```
631 leet = function(head)
632   for line in nodetraverseid(Hhead,head) do
633     for i in nodetraverseid(GLYPH,line.head) do
634       if not leetspeak_onlytext or
635         node.has_attribute(i,luatexbase.attributes.leetattr)
636       then
637         if leettable[i.char] then
638           i.char = leettable[i.char]
639         end
640       end
641     end
642   end
643   return head
644 end
```

10.9 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>

10.9.1 setup of variables

```
645 local letterspace_glue = nodenew(nodeid"glue")
646 local letterspace_spec = nodenew(nodeid"glue_spec")
647 local letterspace_pen = nodenew(nodeid"penalty")
648
649 letterspace_spec.width = tex.sp"0pt"
650 letterspace_spec.stretch = tex.sp"2pt"
651 letterspace_glue.spec = letterspace_spec
652 letterspace_pen.penalty = 10000
```

10.9.2 function implementation

```
653 letterspaceadjust = function(head)
654   for glyph in nodetraverseid(nodeid"glyph", head) do
655     if glyph.prev and (glyph.prev.id == nodeid"glyph" or glyph.prev.id == nodeid"disc") then
656       local g = nodecopy(letterspace_glue)
657       nodeinsertbefore(head, glyph, g)
658       nodeinsertbefore(head, g, nodecopy(letterspace_pen))
659     end
660   end
661   return head
662 end
```

10.9.3 textletterspaceadjust

The `\text...`-version of `letterspaceadjust`. Just works, without the need to call `\letterspaceadjust` globally or anything else. Just put the `\textletterspaceadjust` around the part of text you want the function to work on. Might have problems with surrounding spacing, take care!

```
663 textletterspaceadjust = function(head)
664   for glyph in node.traverse_id(node.id"glyph", head) do
665     if node.has_attribute(glyph, luatexbase.attributes.letterspaceadjustattr) then
666       if glyph.prev and (glyph.prev.id == node.id"glyph" or glyph.prev.id == node.id"disc") then
667         local g = node.copy(letterspace_glue)
668         node.insert_before(head, glyph, g)
669         node.insert_before(head, g, node.copy(letterspace_pen))
670       end
671     end
672   end
673   luatexbase.remove_from_callback("pre_linebreak_filter", "textletterspaceadjust")
674   return head
675 end
```

10.10 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.

```

676 matrixize = function(head)
677   x = {}
678   s = nodenew(nodeid"disc")
679   for n in nodetraverseid(nodeid"glyph",head) do
680     j = n.char
681     for m = 0,7 do -- stay ASCII for now
682       x[7-m] = nodecopy(n) -- to get the same font etc.
683     end
684     if (j / (2^(7-m)) < 1) then
685       x[7-m].char = 48
686     else
687       x[7-m].char = 49
688       j = j-(2^(7-m))
689     end
690     nodeinsertbefore(head,n,x[7-m])
691     nodeinsertafter(head,x[7-m],nodecopy(s))
692   end
693   noderemove(head,n)
694 end
695 return head
696 end

```

10.11 pancakenize

```

697 local separator      = string.rep("=", 28)
698 local texiowrite_nl = texio.write_nl
699 pancaketext = function()
700   texiowrite_nl("Output written on "..tex.jobname.." pdf ("..status.total_pages.." chicken, "..eg
701   texiowrite_nl(" ")
702   texiowrite_nl(separator)
703   texiowrite_nl("Soo ... you decided to use \\pancakenize.")
704   texiowrite_nl("That means you owe me a pancake!")
705   texiowrite_nl(" ")
706   texiowrite_nl("(This goes by document, not compilation.)")
707   texiowrite_nl(separator.."\\n\\n")
708   texiowrite_nl("Looking forward for my pancake! :)")
709   texiowrite_nl("\\n\\n")
710 end

```

10.12 randomerror

10.13 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.

```

711 randomfontslower = 1
712 randomfontsupper = 0

```

```

713 %
714 randomfonts = function(head)
715   local rfub
716   if randomfontsupper > 0 then -- fixme: this should be done only once, no? Or at every paragraph
717     rfub = randomfontsupper -- user-specified value
718   else
719     rfub = font.max() -- or just take all fonts
720   end
721   for line in nodetraverseid(Hhead,head) do
722     for i in nodetraverseid(GLYPH,line.head) do
723       if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattrib) then
724         i.font = math.random(randomfontslower,rfub)
725       end
726     end
727   end
728   return head
729 end

```

10.14 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```

730 uclcratio = 0.5 -- ratio between uppercase and lower case
731 randomuclc = function(head)
732   for i in nodetraverseid(37,head) do
733     if not(randomuclc_onlytext) or node.has_attribute(i,luatexbase.attributes.randuclcattrib) then
734       if math.random() < uclcratio then
735         i.char = tex.uccode[i.char]
736       else
737         i.char = tex.lccode[i.char]
738       end
739     end
740   end
741   return head
742 end

```

10.15 randomchars

```

743 randomchars = function(head)
744   for line in nodetraverseid(Hhead,head) do
745     for i in nodetraverseid(GLYPH,line.head) do
746       i.char = math.floor(math.random()*512)
747     end
748   end
749   return head
750 end

```


10.16 randomcolor and rainbowcolor

10.16.1 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.

```
751 randomcolor_grey = false
752 randomcolor_onlytext = false --switch between local and global colorization
753 rainbowcolor = false
754
755 grey_lower = 0
756 grey_upper = 900
757
758 Rgb_lower = 1
759 rGb_lower = 1
760 rgB_lower = 1
761 Rgb_upper = 254
762 rGb_upper = 254
763 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.

```
764 rainbow_step = 0.005
765 rainbow_Rgb = 1-rainbow_step -- we start in the red phase
766 rainbow_rGb = rainbow_step -- values x must always be  $0 < x < 1$ 
767 rainbow_rgB = rainbow_step
768 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.

```
769 randomcolorstring = function()
770   if randomcolor_grey then
771     return (0.001*math.random(grey_lower, grey_upper)).. " g"
772   elseif rainbowcolor then
773     if rainind == 1 then -- red
774       rainbow_rGb = rainbow_rGb + rainbow_step
775       if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
776     elseif rainind == 2 then -- yellow
777       rainbow_Rgb = rainbow_Rgb - rainbow_step
778       if rainbow_Rgb <= rainbow_step then rainind = 3 end
779     elseif rainind == 3 then -- green
780       rainbow_rgB = rainbow_rgB + rainbow_step
781       rainbow_rGb = rainbow_rGb - rainbow_step
782       if rainbow_rGb <= rainbow_step then rainind = 4 end
783     elseif rainind == 4 then -- blue
784       rainbow_Rgb = rainbow_Rgb + rainbow_step
785       if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
786     else -- purple
787       rainbow_rgB = rainbow_rgB - rainbow_step
```

```

788     if rainbow_rgb <= rainbow_step then rainind = 1 end
789     end
790     return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rgB.." rg"
791 else
792     Rgb = math.random(Rgb_lower,Rgb_upper)/255
793     rGb = math.random(rGb_lower,rGb_upper)/255
794     rgB = math.random(rgB_lower,rgB_upper)/255
795     return Rgb.." "..rGb.." "..rgB.." .." rg"
796 end
797 end

```

10.16.2 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. Otherwise, all glyphs are taken.

```

798 randomcolor = function(head)
799   for line in nodetraverseid(0,head) do
800     for i in nodetraverseid(37,line.head) do
801       if not(randomcolor_onlytext) or
802         (node.has_attribute(i,luatexbase.attributes.randcolorattr))
803       then
804         color_push.data = randomcolorstring() -- color or grey string
805         line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
806         nodeinsertafter(line.head,i,nodecopy(color_pop))
807       end
808     end
809   end
810   return head
811 end

```

10.17 randomerror

```
812 %
```

10.18 rickroll

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

10.19 substitutewords

This function is one of the rather usefull ones of this package. It replaces each occurrence of one word by another word, which both are specified by the user. So nothing random or funny, but a real serious function! There are three levels for this function: At user-level, the user just specifies two strings that are passed to the function `addtosubstitutions`. This is needed as the `#` has a special meaning both in \TeX s definitions and in Lua. In this second step, the list of substitutions is just extended, and the real work is

done by the function `substituteword` which is registered in the `process_input_buffer` callback. Once the substitution list is built, the rest is very simple: We just use `gsub` to substitute, do this for every item in the list, and that's it.

```

813 substitutewords_strings = {}
814
815 addtosubstitutions = function(input,output)
816   substitutewords_strings[#substitutewords_strings + 1] = {}
817   substitutewords_strings[#substitutewords_strings][1] = input
818   substitutewords_strings[#substitutewords_strings][2] = output
819 end
820
821 substitutewords = function(head)
822   for i = 1,#substitutewords_strings do
823     head = string.gsub(head,substitutewords_strings[i][1],substitutewords_strings[i][2])
824   end
825   return head
826 end

```

10.20 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.

```

827 tabularasa_onlytext = false
828
829 tabularasa = function(head)
830   local s = nodenew(nodeid"kern")
831   for line in nodetraverseid(nodeid"hlist",head) do
832     for n in nodetraverseid(nodeid"glyph",line.head) do
833       if not(tabularasa_onlytext) or node.has_attribute(n,luatexbase.attributes.tabularasaattr) then
834         s.kern = n.width
835         nodeinsertafter(line.list,n,nodecopy(s))
836         line.head = noderemove(line.list,n)
837       end
838     end
839   end
840   return head
841 end

```

10.21 uppercasecolor

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

```

842 uppercasecolor_onlytext = false
843
844 uppercasecolor = function (head)
845   for line in nodetraverseid(Hhead,head) do
846     for upper in nodetraverseid(GLYPH,line.head) do

```

```

847     if not(uppercasecolor_onlytext) or node.has_attribute(upper,luatexbase.attributes.uppercase
848     if (((upper.char > 64) and (upper.char < 91)) or
849         ((upper.char > 57424) and (upper.char < 57451))) then -- for small caps! nice
850         color_push.data = randomcolorstring() -- color or grey string
851         line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
852         nodeinsertafter(line.head,upper,nodecopy(color_pop))
853     end
854 end
855 end
856 end
857 return head
858 end

```

10.22 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!

10.22.1 colorstretch – preliminaries

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

```

859 keeptext = true
860 colorexpansion = true
861
862 colorstretch_coloroffset = 0.5
863 colorstretch_colorange = 0.5
864 chickenize_rule_bad_height = 4/5 -- height and depth of the rules
865 chickenize_rule_bad_depth = 1/5
866
867
868 colorstretchnumbers = true
869 drawstretchthreshold = 0.1
870 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 (`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.

```

871 colorstretch = function (head)
872   local f = font.getfont(font.current()).characters
873   for line in nodetraverseid(Hhead,head) do
874     local rule_bad = nodenew(RULE)
875
876     if colorexpanansion then -- if also the font expansion should be shown
877       local g = line.head
878       while not(g.id == 37) do
879         g = g.next
880       end
881       exp_factor = g.width / f[g.char].width
882       exp_color = colorstretch_coloroffset + (1-exp_factor)*10 .. " g"
883       rule_bad.width = 0.5*line.width -- we need two rules on each line!
884     else
885       rule_bad.width = line.width -- only the space expansion should be shown, only one rule
886     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.

```

887   rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bet
888   rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth
889
890   local glue_ratio = 0
891   if line.glue_order == 0 then
892     if line.glue_sign == 1 then
893       glue_ratio = colorstretch_colorrang * math.min(line.glue_set,1)
894     else
895       glue_ratio = -colorstretch_colorrang * math.min(line.glue_set,1)
896     end
897   end
898   color_push.data = colorstretch_coloroffset + glue_ratio .. " g"
899

```

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

```

900 -- set up output
901   local p = line.head
902
903   -- a rule to immitate kerning all the way back
904   local kern_back = nodenew(RULE)
905   kern_back.width = -line.width
906
907   -- if the text should still be displayed, the color and box nodes are inserted additionally
908   -- and the head is set to the color node
909   if keptext then
910     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
911   else
912     node.flush_list(p)

```

```

913     line.head = nodecopy(color_push)
914 end
915 nodeinsertafter(line.head,line.head,rule_bad) -- then the rule
916 nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
917 tmpnode = nodeinsertafter(line.head,line.head.next.next,kern_back)
918
919 -- then a rule with the expansion color
920 if colorexansion then -- if also the stretch/shrink of letters should be shown
921     color_push.data = exp_color
922     nodeinsertafter(line.head,tmpnode,nodecopy(color_push))
923     nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
924     nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
925 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.

```

926 if colorstretchnumbers then
927     j = 1
928     glue_ratio_output = {}
929     for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
930         local char = unicode.utf8.char(s)
931         glue_ratio_output[j] = nodenew(37,1)
932         glue_ratio_output[j].font = font.current()
933         glue_ratio_output[j].char = s
934         j = j+1
935     end
936     if math.abs(glue_ratio) > drawstretchthreshold then
937         if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
938         else color_push.data = "0 0.99 0 rg" end
939     else color_push.data = "0 0 0 rg"
940     end
941
942     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
943     for i = 1,math.min(j-1,7) do
944         nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])
945     end
946     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
947 end -- end of stretch number insertion
948 end
949 return head
950 end

```

dubstepize

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

951

scorpionize

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

```
952 function scorpionize_color(head)
953   color_push.data = ".35 .55 .75 rg"
954   nodeinsertafter(head,head,nodecopy(color_push))
955   nodeinsertafter(head,node.tail(head),nodecopy(color_pop))
956   return head
957 end
```

10.23 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`.

10.23.1 zebranize – preliminaries

```
958 zebracolorarray = {}
959 zebracolorarray_bg = {}
960 zebracolorarray[1] = "0.1 g"
961 zebracolorarray[2] = "0.9 g"
962 zebracolorarray_bg[1] = "0.9 g"
963 zebracolorarray_bg[2] = "0.1 g"
```

10.23.2 zebranize – the function

This code has to be revisited, it is ugly.

```
964 function zebranize(head)
965   zebracolor = 1
966   for line in nodetraverseid(nodeid"hhead",head) do
967     if zebracolor == #zebracolorarray then zebracolor = 0 end
968     zebracolor = zebracolor + 1
969     color_push.data = zebracolorarray[zebracolor]
970     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
971     for n in nodetraverseid(nodeid"glyph",line.head) do
972       if n.next then else
```

```

973         nodeinsertafter(line.head,n,nodecopy(color_pull))
974     end
975 end
976
977 local rule_zebra = nodenew(RULE)
978 rule_zebra.width = line.width
979 rule_zebra.height = tex.baselineskip.width*4/5
980 rule_zebra.depth = tex.baselineskip.width*1/5
981
982 local kern_back = nodenew(RULE)
983 kern_back.width = -line.width
984
985 color_push.data = zebracolorarray_bg[zebracolor]
986 line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
987 line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
988 nodeinsertafter(line.head,line.head,kern_back)
989 nodeinsertafter(line.head,line.head,rule_zebra)
990 end
991 return (head)
992 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!

```

993 --
994 function pdf_print (...)
995   for _, str in ipairs({...}) do
996     pdf.print(str .. " ")
997   end
998   pdf.print("\string\n")
999 end
1000
1001 function move (p)
1002   pdf_print(p[1],p[2],"m")
1003 end
1004
1005 function line (p)
1006   pdf_print(p[1],p[2],"l")
1007 end
1008
1009 function curve(p1,p2,p3)
1010   pdf_print(p1[1], p1[2],
1011             p2[1], p2[2],
1012             p3[1], p3[2], "c")
1013 end
1014
1015 function close ()
1016   pdf_print("h")
1017 end
1018
1019 function linewidth (w)
1020   pdf_print(w,"w")
1021 end
1022
1023 function stroke ()
1024   pdf_print("S")
1025 end
1026 --
1027

```

```

1028 function strictcircle(center,radius)
1029   local left = {center[1] - radius, center[2]}
1030   local lefttop = {left[1], left[2] + 1.45*radius}
1031   local leftbot = {left[1], left[2] - 1.45*radius}
1032   local right = {center[1] + radius, center[2]}
1033   local righttop = {right[1], right[2] + 1.45*radius}
1034   local rightbot = {right[1], right[2] - 1.45*radius}
1035
1036   move (left)
1037   curve (lefttop, righttop, right)
1038   curve (rightbot, leftbot, left)
1039 stroke()
1040 end
1041
1042 function disturb_point(point)
1043   return {point[1] + math.random()*5 - 2.5,
1044           point[2] + math.random()*5 - 2.5}
1045 end
1046
1047 function sloppycircle(center,radius)
1048   local left = disturb_point({center[1] - radius, center[2]})
1049   local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
1050   local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}
1051   local right = disturb_point({center[1] + radius, center[2]})
1052   local righttop = disturb_point({right[1], right[2] + 1.45*radius})
1053   local rightbot = disturb_point({right[1], right[2] - 1.45*radius})
1054
1055   local right_end = disturb_point(right)
1056
1057   move (right)
1058   curve (rightbot, leftbot, left)
1059   curve (lefttop, righttop, right_end)
1060   linewidth(math.random()+0.5)
1061   stroke()
1062 end
1063
1064 function sloppyline(start,stop)
1065   local start_line = disturb_point(start)
1066   local stop_line = disturb_point(stop)
1067   start = disturb_point(start)
1068   stop = disturb_point(stop)
1069   move(start) curve(start_line,stop_line,stop)
1070   linewidth(math.random()+0.5)
1071   stroke()
1072 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 Do's

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

countglyphs should be extended to count anything the user wants to count

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

pancakenize should do something funny.

chickenize should differ between character and punctuation.

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

chickenmath chickenization of math mode

14 Literature

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

- Lua_T_EX 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 Lua_T_EX 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 ...