

*»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 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 Lua \TeX features, so don't even try to use it with any other \TeX flavour. The package is tested under plain Lua \TeX and Lua \LaTeX . If you tried using it with Con \TeX t, please share your experience, I will gladly try to make it compatible!

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

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

less useful functions

countglyphs	counts the number of glyphs in the whole document
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” (or user-adjustable words)
gutenbergize	deletes every quote and footnotes
hammertime	U can't touch this!
kernmanipulate	manipulates the kerning (tbi)
matrixize	replaces every glyph by its ASCII value in binary code
randomerror	just throws random (La)TeX errors at random times
randomfonts	changes the font randomly between every letter
randomchars	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	6
2.3	\text-Versions	7
2.4	Lua functions	7
3	Options – How to Adjust It	7
II	Tutorial	9
4	Lua code	9
5	callbacks	9
5.1	How to use a callback	10
6	Nodes	10
7	Other things	11
III	Implementation	12
8	TeX file	12
9	TeX package	19
9.1	Definition of User-Level Macros	19
10	Lua Module	19
10.1	chickenize	20
10.2	countglyphs	22
10.3	gutenbergize	23
10.3.1	gutenbergize – preliminaries	23
10.3.2	gutenbergize – the function	23
10.4	hammertime	24
10.5	itsame	24
10.6	kernmanipulate	25
10.7	leetspeak	26
10.8	letterspaceadjust	26
10.8.1	setup of variables	27
10.8.2	function implementation	27

10.8.3 textletterspaceadjust	27
10.9 matrixize	27
10.10 pancakenize	28
10.11 randomerror	28
10.12 randomfonts	28
10.13 randomuclc	29
10.14 randomchars	29
10.15 randomcolor and rainbowcolor	30
10.15.1 randomcolor – preliminaries	30
10.15.2 randomcolor – the function	31
10.16 randomerror	31
10.17 rickroll	31
10.18 substitutewords	31
10.19 tabularasa	32
10.20 uppercasecolor	32
10.21 colorstretch	33
10.21.1 colorstretch – preliminaries	33
10.22 zebranize	36
10.22.1 zebranize – preliminaries	36
10.22.2 zebranize – the function	36
11 Drawing	38
12 Known Bugs	40
13 To Do's	40
14 Literature	40
15 Thanks	40

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](#).

`\countglyphs` Counts every printed character that appeared in anything that is a paragraph. Which is quite everything, in fact, *except* 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.³

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

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

`\hammertime` STOP! — Hammertime!

`\uppercasecolor` Makes every uppercase character in the input colored. At the moment, the color is randomized over the full rgb scale, but that will be adjustable once options are well implemented.

³If you have a nice implementation idea, I'd love to include this!

- `\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.
- `\pancakelize` This is a dummy command that does nothing. However, every time you use it, you owe a pancake to the package author. You can either send it via mail or bring it to some (local) \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 it's functionality. So once you used `\chickenize`, it will chickenize the whole document up to `\unchickenize`. However, the paragraph in which `\unchickenize` appears, will *not* be chickenized. The same is true for all other manipulations. Take care that you don't `\un`-anything befor activating it, as this will result in an error.⁴

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.

⁴Which is so far not catchable due to missing functionality in `luatexbase`.

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 have⁵ 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.⁶

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.

⁵If they don't have, I did miss that, sorry. Please inform me about such cases.

⁶On 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`!

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.

colorexpanansion = **<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 Lua_T_EX manual! Then, you have to remove the `if n.char` condition on the third line of the listing, because glue nodes lack a `.char` field. If everything works, you should have an input consisting of only one long word. Congratulations!

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

7 Other things

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

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

The namespace of the `chickenize` package is *not* consistent. Please don't take anything here as an example for good Lua coding, for good \TeX ing or even for good Lua_T_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 \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 \LaTeX '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\BEClerialize{
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\chickenize{
28   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize")}
29   luatexbase.add_to_callback("start_page_number",
30     function() texio.write("[\"..status.total_pages) end \", "cstartpage")
31   luatexbase.add_to_callback("stop_page_number",
```

```

32     function() texio.write(" chickens]") end,"cstoppage")
33 %
34     luatexbase.add_to_callback("stop_run",nicetext,"a nice text")
35 }
36 }
37 \def\unchickenize{
38   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","chickenize")}
39   luatexbase.remove_from_callback("start_page_number","cstartpage")
40   luatexbase.remove_from_callback("stop_page_number","cstoppage")}}
41
42 \def\coffeestainize{ %% to be implemented.
43   \directlua{}}
44 \def\uncoffeestainize{
45   \directlua{}}
46
47 \def\colorstretch{
48   \directlua{luatexbase.add_to_callback("post_linebreak_filter",colorstretch,"stretch_expansion")}
49 \def\uncolorstretch{
50   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","stretch_expansion")}}
51
52 \def\countglyphs{
53   \directlua{glyphnumber = 0
54             luatexbase.add_to_callback("post_linebreak_filter",countglyphs,"countglyphs")
55             luatexbase.add_to_callback("stop_run",printglyphnumber,"printglyphnumber")
56   }
57 }
58
59 \def\dosomethingfunny{
60   %% should execute one of the "funny" commands, but randomly. So every compilation is complete
61 }
62
63 \def\dubstepenize{
64   \chickenize
65   \directlua{
66     chickenstring[1] = "WOB"
67     chickenstring[2] = "WOB"
68     chickenstring[3] = "WOB"
69     chickenstring[4] = "BROOOAR"
70     chickenstring[5] = "WHEE"
71     chickenstring[6] = "WOB WOB WOB"
72     chickenstring[7] = "WAAAAAAAAAH"
73     chickenstring[8] = "duhduh duhduh duh"
74     chickenstring[9] = "BEEEEEEEEEW"
75     chickenstring[10] = "DEEEEEEEEEW"
76     chickenstring[11] = "EEEEEW"
77     chickenstring[12] = "boop"

```

```

78     chickenstring[13] = "buhdee"
79     chickenstring[14] = "bee bee"
80     chickenstring[15] = "BZZZRRRRRRR000000AAAAA"
81
82     chickenizefraction = 1
83 }
84 }
85 \let\dubstepize\dubstepenize
86
87 \def\gutenbergize{ %% makes only sense when using LaTeX
88   \AtBeginDocument{
89     \let\grqq\relax\let\glqq\relax
90     \let\frqq\relax\let\flqq\relax
91     \let\grq\relax\let\glq\relax
92     \let\frq\relax\let\flq\relax
93 %
94     \gdef\footnote##1{}
95     \gdef\cite##1{}\gdef\parencite##1{}
96     \gdef\Cite##1{}\gdef\Parencite##1{}
97     \gdef\cites##1{}\gdef\parencites##1{}
98     \gdef\Cites##1{}\gdef\Parencites##1{}
99     \gdef\footcite##1{}\gdef\footcitetext##1{}
100    \gdef\footcites##1{}\gdef\footcitetexts##1{}
101    \gdef\textcite##1{}\gdef\Textcite##1{}
102    \gdef\textcites##1{}\gdef\Textcites##1{}
103    \gdef\smartcites##1{}\gdef\Smartcites##1{}
104    \gdef\supercite##1{}\gdef\supercites##1{}
105    \gdef\autocite##1{}\gdef\Autocite##1{}
106    \gdef\autocites##1{}\gdef\Autocites##1{}
107    %% many, many missing ... maybe we need to tackle the underlying mechanism?
108  }
109  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",gutenbergize_rq,"gutenbergize_rq")}
110 }
111
112 \def\hammertime{
113   \global\let\n\relax
114   \directlua{hammerfirst = true
115             luatexbase.add_to_callback("pre_linebreak_filter",hammertime,"hammertime")}}
116 \def\unhammertime{
117   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","hammertime")}}
118
119 % \def\itsame{
120 %   \directlua{drawmario}} %% does not exist
121
122 \def\kernmanipulate{
123   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",kernmanipulate,"kernmanipulate")}}

```

```

124 \def\unkernmanipulate{
125   \directlua{luaexbase.remove_from_callback("pre_linebreak_filter",kernmanipulate)}}
126
127 \def\leetspeak{
128   \directlua{luaexbase.add_to_callback("post_linebreak_filter",leet,"1337")}}
129 \def\unleetspeak{
130   \directlua{luaexbase.remove_from_callback("post_linebreak_filter","1337")}}
131
132 \def\letterspaceadjust{
133   \directlua{luaexbase.add_to_callback("pre_linebreak_filter",letterspaceadjust,"letterspaceadjust")}}
134 \def\unletterspaceadjust{
135   \directlua{luaexbase.remove_from_callback("pre_linebreak_filter","letterspaceadjust")}}
136
137 \def\listallcommands{
138   \directlua{
139     for name in pairs(tex.hashtokens()) do
140       print(name)
141     end}
142 }
143
144 \let\stealsheep\letterspaceadjust      %% synonym in honor of Paul
145 \let\unstealsheep\unletterspaceadjust
146 \let\returnsheep\unletterspaceadjust
147
148 \def\matrixize{
149   \directlua{luaexbase.add_to_callback("pre_linebreak_filter",matrixize,"matrixize")}}
150 \def\unmatrixize{
151   \directlua{luaexbase.remove_from_callback("pre_linebreak_filter",matrixize)}}
152
153 \def\milkcow{      %% FIXME %% to be implemented
154   \directlua{}}
155 \def\unmilkcow{
156   \directlua{}}
157
158 \def\pancakenize{
159   \directlua{luaexbase.add_to_callback("stop_run",pancaketext,"pancaketext")}}
160
161 \def\rainbowcolor{
162   \directlua{luaexbase.add_to_callback("post_linebreak_filter",randomcolor,"rainbowcolor")
163     rainbowcolor = true}}
164 \def\unrainbowcolor{
165   \directlua{luaexbase.remove_from_callback("post_linebreak_filter","rainbowcolor")
166     rainbowcolor = false}}
167 \let\nyanize\rainbowcolor
168 \let\unnyanize\unrainbowcolor
169

```

```

170 \def\randomcolor{
171   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"randomcolor")}}
172 \def\unrandomcolor{
173   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomcolor")}}
174
175 \def\randomerror{ %% FIXME
176   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomerror,"randomerror")}}
177 \def\unrandomerror{ %% FIXME
178   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomerror")}}
179
180 \def\randomfonts{
181   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomfonts,"randomfonts")}}
182 \def\unrandomfonts{
183   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomfonts")}}
184
185 \def\randomuclc{
186   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",randomuclc,"randomuclc")}}
187 \def\unrandomuclc{
188   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","randomuclc")}}
189
190 \def\scorpionize{
191   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",scorpionize_color,"scorpionize_color")}}
192 \def\unscorpionize{
193   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","scorpionize_color")}}
194
195 \def\spankmonkey{ %% to be implemented
196   \directlua{}}
197 \def\unspankmonkey{
198   \directlua{}}
199
200 \def\substitutewords{
201   \directlua{luatexbase.add_to_callback("process_input_buffer",substitutewords,"substitutewords")}}
202 \def\unsubstitutewords{
203   \directlua{luatexbase.remove_from_callback("process_input_buffer","substitutewords")}}
204
205 \def\addtosubstitutions#1#2{
206   \directlua{addtosubstitutions("#1","#2")}}
207 }
208
209 \def\tabularasa{
210   \directlua{luatexbase.add_to_callback("post_linebreak_filter",tabularasa,"tabularasa")}}
211 \def\untabularasa{
212   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","tabularasa")}}
213
214 \def\uppercasecolor{
215   \directlua{luatexbase.add_to_callback("post_linebreak_filter",uppercasecolor,"uppercasecolor")}}

```



```

216 \def\unuppercasecolor{
217   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","uppercasecolor")}}
218
219 \def\zebranize{
220   \directlua{luatexbase.add_to_callback("post_linebreak_filter",zebranize,"zebranize")}}
221 \def\unzebranize{
222   \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.

```

223 \newluatexattribute\leetattr
224 \newluatexattribute\letterspaceadjustattr
225 \newluatexattribute\randcolorattr
226 \newluatexattribute\randfontsattrib
227 \newluatexattribute\randucclcattrib
228 \newluatexattribute\tabularasaaattr
229 \newluatexattribute\uppercasecolorattr
230
231 \long\def\textleetspeak#1%
232   {\setluatexattribute\leetattr{42}#1\unsetluatexattribute\leetattr}
233
234 \long\def\textletterspaceadjust#1{
235   \setluatexattribute\letterspaceadjustattr{42}#1\unsetluatexattribute\letterspaceadjustattr
236   \directlua{
237     if (textletterspaceadjustactive) then else % -- if already active, do nothing
238       luatexbase.add_to_callback("pre_linebreak_filter",textletterspaceadjust,"textletterspaceadj
239     end
240     textletterspaceadjustactive = true           % -- set to active
241   }
242 }
243 \let\textlsa\textletterspaceadjust
244
245 \long\def\textrandomcolor#1%
246   {\setluatexattribute\randcolorattr{42}#1\unsetluatexattribute\randcolorattr}
247 \long\def\textrandomfontsa#1%
248   {\setluatexattribute\randfontsattrib{42}#1\unsetluatexattribute\randfontsattrib}
249 \long\def\textrandomfontsc#1%
250   {\setluatexattribute\randfontsattrib{42}#1\unsetluatexattribute\randfontsattrib}
251 \long\def\textrandomucclc#1%
252   {\setluatexattribute\randucclcattrib{42}#1\unsetluatexattribute\randucclcattrib}
253 \long\def\texttabularasa#1%
254   {\setluatexattribute\tabularasaaattr{42}#1\unsetluatexattribute\tabularasaaattr}
255 \long\def\textuppercasecolor#1%
256   {\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.

```
257 \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.

```
258 \long\def\luadraw#1#2{%
259   \vbox to #1bp{%
260     \vfil
261     \luatexlatalua{pdf_print("q") #2 pdf_print("Q")}%
262   }%
263 }
264 \long\def\drawchicken{
265   \luadraw{90}{
266     kopf = {200,50} % Kopfmitte
267     kopf_rad = 20
268
269     d = {215,35} % Halsansatz
270     e = {230,10} %
271
272     korper = {260,-10}
273     korper_rad = 40
274
275     bein11 = {260,-50}
276     bein12 = {250,-70}
277     bein13 = {235,-70}
278
279     bein21 = {270,-50}
280     bein22 = {260,-75}
281     bein23 = {245,-75}
282
283     schnabel_oben = {185,55}
284     schnabel_vorne = {165,45}
285     schnabel_unten = {185,35}
286
287     flugel_vorne = {260,-10}
288     flugel_unten = {280,-40}
289     flugel_hinten = {275,-15}
290
291     sloppyline(kopf,kopf_rad)
292     sloppyline(d,e)
293     sloppyline(korper,korper_rad)
294     sloppyline(bein11,bein12) sloppyline(bein12,bein13)
295     sloppyline(bein21,bein22) sloppyline(bein22,bein23)
296     sloppyline(schnabel_vorne,schnabel_oben) sloppyline(schnabel_vorne,schnabel_unten)
297     sloppyline(flugel_vorne,flugel_unten) sloppyline(flugel_hinten,flugel_unten)
298   }
299 }
```

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.

```
300 \ProvidesPackage{chickenize}%
301 [2012/09/16 v0.1a chickenize package]
302 \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! :)

```
303 \iffalse
304 \DeclareDocumentCommand\includegraphics{O{m}}{
305     \fbox{Chicken} %% actually, I'd love to draw an MP graph showing a chicken ...
306 }
307 %%% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
308 %% So far, you have to load pgfplots yourself.
309 %% As it is a mighty package, I don't want the user to force loading it.
310 \NewDocumentCommand\balmerpeak{G{}O{-4cm}}{
311 %% to be done using Lua drawing.
312 }
313 \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.

```
314
315 local nodenew = node.new
316 local nodecopy = node.copy
317 local nodeinsertbefore = node.insert_before
318 local nodeinsertafter = node.insert_after
319 local noderemove = node.remove
320 local nodeid = node.id
321 local nodetraverseid = node.traverse_id
322 local nodeslide = node.slide
323
324 Hhead = nodeid("hhead")
```

```

325 RULE = nodeid("rule")
326 GLUE = nodeid("glue")
327 WHAT = nodeid("whatsit")
328 COL = node.subtype("pdf_colorstack")
329 GLYPH = nodeid("glyph")

```

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

```

330 color_push = nodenew(WHAT,COL)
331 color_pop = nodenew(WHAT,COL)
332 color_push.stack = 0
333 color_pop.stack = 0
334 color_push.cmd = 1
335 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.

```

336 chicken_pagenumbers = true
337
338 chickenstring = {}
339 chickenstring[1] = "chicken" -- chickenstring is a table, please remeber this!
340
341 chickenizefraction = 0.5
342 -- set this to a small value to fool somebody, or to see if your text has been read carefully. Th
343 chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing your
344
345 local tbl = font.getfont(font.current())
346 local space = tbl.parameters.space
347 local shrink = tbl.parameters.space_shrink
348 local stretch = tbl.parameters.space_stretch
349 local match = unicode.utf8.match
350 chickenize_ignore_word = false

```

The function chickenize_real_stuff is started once the beginning of a to-be-substituted word is found.

```

351 chickenize_real_stuff = function(i,head)
352     while ((i.next.id == 37) or (i.next.id == 11) or (i.next.id == 7) or (i.next.id == 0)) do --
353         i.next = i.next.next
354     end
355
356     chicken = {} -- constructing the node list.
357
358 -- Should this be done only once? No, otherwise we lose the freedom to change the string in-docum
359 -- But it could be done only once each paragraph as in-paragraph changes are not possible!
360
361     chickenstring_tmp = chickenstring[math.random(1,#chickenstring)]

```

```

362     chicken[0] = nodenew(37,1) -- only a dummy for the loop
363     for i = 1,string.len(chickenstring_tmp) do
364         chicken[i] = nodenew(37,1)
365         chicken[i].font = font.current()
366         chicken[i-1].next = chicken[i]
367     end
368
369     j = 1
370     for s in string.utfvalues(chickenstring_tmp) do
371         local char = unicode.utf8.char(s)
372         chicken[j].char = s
373         if match(char,"%s") then
374             chicken[j] = nodenew(10)
375             chicken[j].spec = nodenew(47)
376             chicken[j].spec.width = space
377             chicken[j].spec.shrink = shrink
378             chicken[j].spec.stretch = stretch
379         end
380         j = j+1
381     end
382
383     nodeslide(chicken[1])
384     lang.hyphenate(chicken[1])
385     chicken[1] = node.kerning(chicken[1]) -- FIXME: does not work
386     chicken[1] = node.ligaturing(chicken[1]) -- dito
387
388     nodeinsertbefore(head,i,chicken[1])
389     chicken[1].next = chicken[2] -- seems to be necessary ... to be fixed
390     chicken[string.len(chickenstring_tmp)].next = i.next
391
392     -- shift lowercase latin letter to uppercase if the original input was an uppercase
393     if (chickenize_capital and (chicken[1].char > 96 and chicken[1].char < 123)) then
394         chicken[1].char = chicken[1].char - 32
395     end
396
397     return head
398 end
399
400 chickenize = function(head)
401     for i in nodetraverseid(37,head) do --find start of a word
402         if (chickenize_ignore_word == false) then -- normal case: at the beginning of a word, we jump
403             if (i.char > 64 and i.char < 91) then chickenize_capital = true else chickenize_capital = false
404             head = chickenize_real_stuff(i,head)
405         end
406     end
407 -- At the end of the word, the ignoring is reset. New chance for everyone.

```

```

408     if not((i.next.id == 37) or (i.next.id == 7) or (i.next.id == 22) or (i.next.id == 11)) then
409         chickenize_ignore_word = false
410     end
411
412 -- And the random determination of the chickenization of the next word:
413     if math.random() > chickenizefraction then
414         chickenize_ignore_word = true
415     elseif chickencount then
416         chicken_substitutions = chicken_substitutions + 1
417     end
418 end
419 return head
420 end
421

```

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)

```

422 local separator      = string.rep("=", 28)
423 local texiowrite_nl = texio.write_nl
424 nicetext = function()
425     texiowrite_nl("Output written on "..tex.jobname.."pdf ("..status.total_pages.." chicken,".." e
426     texiowrite_nl(" ")
427     texiowrite_nl(separator)
428     texiowrite_nl("Hello my dear user,")
429     texiowrite_nl("good job, now go outside and enjoy the world!")
430     texiowrite_nl(" ")
431     texiowrite_nl("And don't forget to feed your chicken!")
432     texiowrite_nl(separator .. "\n")
433     if chickencount then
434         texiowrite_nl("There were "..chicken_substitutions.." substitutions made.")
435     texiowrite_nl(separator)
436     end
437 end

```

10.2 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.

```

438 countglyphs = function(head)
439     for line in nodetraverseid(0,head) do
440         for glyph in nodetraverseid(37,line.head) do
441             glyphnumber = glyphnumber + 1
442         end

```

```

443 end
444 return head
445 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? ...

```

446 printglyphnumber = function()
447   texiowrite_nl("Number of glyphs in this document: "..glyphnumber)
448 end

```

10.3 guttenbergenize

A function in honor of the German politician Gutenberg.⁷ Please do *not* confuse him with the grand master Gutenberg!

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

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

10.3.1 guttenbergenize – preliminaries

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

```

449 local quotestrings = {
450   [171] = true, [172] = true,
451   [8216] = true, [8217] = true, [8218] = true,
452   [8219] = true, [8220] = true, [8221] = true,
453   [8222] = true, [8223] = true,
454   [8248] = true, [8249] = true, [8250] = true,
455 }

```

10.3.2 guttenbergenize – the function

```

456 guttenbergenize_rq = function(head)
457   for n in nodetraverseid(nodeid"glyph",head) do
458     local i = n.char
459     if quotestrings[i] then
460       noderemove(head,n)
461     end
462   end
463   return head
464 end

```

⁷Thanks to Jasper for bringing me to this idea!

10.4 hammertime

This is a completely useless function. It just prints STOP! – HAMMERTIME at the beginning of the first paragraph after `\hammertime`, and “U can't touch this” for every following one. As the function writes to the terminal, you have to be sure that your terminal is line-buffered and not block-buffered. Compare the explanation by Taco on the LuaTeX mailing list.⁸

```
465 hammertimedelay = 1.2
466 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
467 hammertime = function(head)
468   if hammerfirst then
469     texiowrite_nl(htime_separator)
470     texiowrite_nl("=====STOP!=====\\n")
471     texiowrite_nl(htime_separator .. "\\n\\n\\n")
472     os.sleep (hammertimedelay*1.5)
473     texiowrite_nl(htime_separator .. "\\n")
474     texiowrite_nl("=====HAMMERTIME=====\\n")
475     texiowrite_nl(htime_separator .. "\\n\\n")
476     os.sleep (hammertimedelay)
477     hammerfirst = false
478   else
479     os.sleep (hammertimedelay)
480     texiowrite_nl(htime_separator)
481     texiowrite_nl("=====U can't touch this!=====\\n")
482     texiowrite_nl(htime_separator .. "\\n\\n")
483     os.sleep (hammertimedelay*0.5)
484   end
485   return head
486 end
```

10.5 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.

```
487 itsame = function()
488 local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
489 color = "1 .6 0"
490 for i = 6,9 do mr(i,3) end
491 for i = 3,11 do mr(i,4) end
492 for i = 3,12 do mr(i,5) end
493 for i = 4,8 do mr(i,6) end
494 for i = 4,10 do mr(i,7) end
495 for i = 1,12 do mr(i,11) end
496 for i = 1,12 do mr(i,12) end
497 for i = 1,12 do mr(i,13) end
498
```

⁸<http://tug.org/pipermail/luatex/2011-November/003355.html>


```

499 color = ".3 .5 .2"
500 for i = 3,5 do mr(i,3) end mr(8,3)
501 mr(2,4) mr(4,4) mr(8,4)
502 mr(2,5) mr(4,5) mr(5,5) mr(9,5)
503 mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
504 for i = 3,8 do mr(i,8) end
505 for i = 2,11 do mr(i,9) end
506 for i = 1,12 do mr(i,10) end
507 mr(3,11) mr(10,11)
508 for i = 2,4 do mr(i,15) end for i = 9,11 do mr(i,15) end
509 for i = 1,4 do mr(i,16) end for i = 9,12 do mr(i,16) end
510
511 color = "1 0 0"
512 for i = 4,9 do mr(i,1) end
513 for i = 3,12 do mr(i,2) end
514 for i = 8,10 do mr(5,i) end
515 for i = 5,8 do mr(i,10) end
516 mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
517 for i = 4,9 do mr(i,12) end
518 for i = 3,10 do mr(i,13) end
519 for i = 3,5 do mr(i,14) end
520 for i = 7,10 do mr(i,14) end
521 end

```

10.6 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.

```

522 chickenkernamount = 0
523 chickeninvertkerning = false
524
525 function kernmanipulate (head)
526   if chickeninvertkerning then -- invert the kerning
527     for n in nodetraverseid(11,head) do
528       n.kern = -n.kern
529     end
530   else -- if not, set it to the given value
531     for n in nodetraverseid(11,head) do
532       n.kern = chickenkernamount
533     end
534   end
535   return head
536 end

```

10.7 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.

```
537 leetspeak_onlytext = false
538 leettable = {
539   [101] = 51, -- E
540   [105] = 49, -- I
541   [108] = 49, -- L
542   [111] = 48, -- O
543   [115] = 53, -- S
544   [116] = 55, -- T
545
546   [101-32] = 51, -- e
547   [105-32] = 49, -- i
548   [108-32] = 49, -- l
549   [111-32] = 48, -- o
550   [115-32] = 53, -- s
551   [116-32] = 55, -- t
552 }
```

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

```
553 leet = function(head)
554   for line in nodetraverseid(Hhead,head) do
555     for i in nodetraverseid(GLYPH,line.head) do
556       if not leetspeak_onlytext or
557         node.has_attribute(i,luatexbase.attributes.leetattr)
558       then
559         if leettable[i.char] then
560           i.char = leettable[i.char]
561         end
562       end
563     end
564   end
565   return head
566 end
```

10.8 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>

10.8.1 setup of variables

```
567 local letterspace_glue = nodenew(nodeid"glue")
568 local letterspace_spec = nodenew(nodeid"glue_spec")
569 local letterspace_pen = nodenew(nodeid"penalty")
570
571 letterspace_spec.width = tex.sp"0pt"
572 letterspace_spec.stretch = tex.sp"2pt"
573 letterspace_glue.spec = letterspace_spec
574 letterspace_pen.penalty = 10000
```

10.8.2 function implementation

```
575 letterspaceadjust = function(head)
576   for glyph in nodetraverseid(nodeid"glyph", head) do
577     if glyph.prev and (glyph.prev.id == nodeid"glyph" or glyph.prev.id == nodeid"disc") then
578       local g = nodecopy(letterspace_glue)
579       nodeinsertbefore(head, glyph, g)
580       nodeinsertbefore(head, g, nodecopy(letterspace_pen))
581     end
582   end
583   return head
584 end
```

10.8.3 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!

```
585 textletterspaceadjust = function(head)
586   for glyph in node.traverse_id(node.id"glyph", head) do
587     if node.has_attribute(glyph, luatexbase.attributes.letterspaceadjustattr) then
588       if glyph.prev and (glyph.prev.id == node.id"glyph" or glyph.prev.id == node.id"disc") then
589         local g = node.copy(letterspace_glue)
590         node.insert_before(head, glyph, g)
591         node.insert_before(head, g, node.copy(letterspace_pen))
592       end
593     end
594   end
595   luatexbase.remove_from_callback("pre_linebreak_filter", "textletterspaceadjust")
596   return head
597 end
```

10.9 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.

```

598 matrixize = function(head)
599   x = {}
600   s = nodenew(nodeid"disc")
601   for n in nodetraverseid(nodeid"glyph",head) do
602     j = n.char
603     for m = 0,7 do -- stay ASCII for now
604       x[7-m] = nodecopy(n) -- to get the same font etc.
605
606       if (j / (2^(7-m)) < 1) then
607         x[7-m].char = 48
608       else
609         x[7-m].char = 49
610         j = j-(2^(7-m))
611       end
612       nodeinsertbefore(head,n,x[7-m])
613       nodeinsertafter(head,x[7-m],nodecopy(s))
614     end
615     noderemove(head,n)
616   end
617   return head
618 end

```

10.10 pancakenize

```

619 local separator      = string.rep("=", 28)
620 local texiowrite_nl = texio.write_nl
621 pancaketext = function()
622   texiowrite_nl("Output written on "..tex.jobname.."pdf ("..status.total_pages.." chicken, ".." eg
623   texiowrite_nl(" ")
624   texiowrite_nl(separator)
625   texiowrite_nl("Soo ... you decided to use \\pancakenize.")
626   texiowrite_nl("That means you owe me a pancake!")
627   texiowrite_nl(" ")
628   texiowrite_nl("(This goes by document, not compilation.)")
629   texiowrite_nl(separator.."\\n\\n")
630   texiowrite_nl("Looking forward for my pancake! :)")
631   texiowrite_nl("\\n\\n")
632 end

```

10.11 randomerror

10.12 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.

```

633 randomfontslower = 1
634 randomfontsupper = 0

```

```

635 %
636 randomfonts = function(head)
637   local rfub
638   if randomfontsupper > 0 then -- fixme: this should be done only once, no? Or at every paragraph
639     rfub = randomfontsupper -- user-specified value
640   else
641     rfub = font.max() -- or just take all fonts
642   end
643   for line in nodetraverseid(Hhead,head) do
644     for i in nodetraverseid(GLYPH,line.head) do
645       if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattrib) then
646         i.font = math.random(randomfontslower,rfub)
647       end
648     end
649   end
650   return head
651 end

```

10.13 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```

652 uclcratio = 0.5 -- ratio between uppercase and lower case
653 randomuclc = function(head)
654   for i in nodetraverseid(37,head) do
655     if not(randomuclc_onlytext) or node.has_attribute(i,luatexbase.attributes.randuclcattrib) then
656       if math.random() < uclcratio then
657         i.char = tex.uccode[i.char]
658       else
659         i.char = tex.lccode[i.char]
660       end
661     end
662   end
663   return head
664 end

```

10.14 randomchars

```

665 randomchars = function(head)
666   for line in nodetraverseid(Hhead,head) do
667     for i in nodetraverseid(GLYPH,line.head) do
668       i.char = math.floor(math.random()*512)
669     end
670   end
671   return head
672 end

```

10.15 randomcolor and rainbowcolor

10.15.1 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.

```
673 randomcolor_grey = false
674 randomcolor_onlytext = false --switch between local and global colorization
675 rainbowcolor = false
676
677 grey_lower = 0
678 grey_upper = 900
679
680 Rgb_lower = 1
681 rGb_lower = 1
682 rgB_lower = 1
683 Rgb_upper = 254
684 rGb_upper = 254
685 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.

```
686 rainbow_step = 0.005
687 rainbow_Rgb = 1-rainbow_step -- we start in the red phase
688 rainbow_rGb = rainbow_step -- values x must always be  $0 < x < 1$ 
689 rainbow_rgB = rainbow_step
690 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.

```
691 randomcolorstring = function()
692   if randomcolor_grey then
693     return (0.001*math.random(grey_lower, grey_upper)).. " g"
694   elseif rainbowcolor then
695     if rainind == 1 then -- red
696       rainbow_rGb = rainbow_rGb + rainbow_step
697       if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
698     elseif rainind == 2 then -- yellow
699       rainbow_Rgb = rainbow_Rgb - rainbow_step
700       if rainbow_Rgb <= rainbow_step then rainind = 3 end
701     elseif rainind == 3 then -- green
702       rainbow_rgB = rainbow_rgB + rainbow_step
703       rainbow_rGb = rainbow_rGb - rainbow_step
704       if rainbow_rGb <= rainbow_step then rainind = 4 end
705     elseif rainind == 4 then -- blue
706       rainbow_Rgb = rainbow_Rgb + rainbow_step
707       if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
708     else -- purple
709       rainbow_rgB = rainbow_rgB - rainbow_step
```

```

710     if rainbow_rgb <= rainbow_step then rainind = 1 end
711     end
712     return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rgB.." rg"
713 else
714     Rgb = math.random(Rgb_lower,Rgb_upper)/255
715     rGb = math.random(rGb_lower,rGb_upper)/255
716     rgB = math.random(rgB_lower,rgB_upper)/255
717     return Rgb.." "..rGb.." "..rgB.." .." rg"
718 end
719 end

```

10.15.2 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.

```

720 randomcolor = function(head)
721   for line in nodetraverseid(0,head) do
722     for i in nodetraverseid(37,line.head) do
723       if not(randomcolor_onlytext) or
724         (node.has_attribute(i,luatexbase.attributes.randcolorattr))
725       then
726         color_push.data = randomcolorstring() -- color or grey string
727         line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
728         nodeinsertafter(line.head,i,nodecopy(color_pop))
729       end
730     end
731   end
732   return head
733 end

```

10.16 randomerror

```

734 %

```

10.17 rickroll

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

10.18 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.

```

735 substitutewords_strings = {}
736
737 addtosubstitutions = function(input,output)
738   substitutewords_strings[#substitutewords_strings + 1] = {}
739   substitutewords_strings[#substitutewords_strings][1] = input
740   substitutewords_strings[#substitutewords_strings][2] = output
741 end
742
743 substitutewords = function(head)
744   for i = 1,#substitutewords_strings do
745     head = string.gsub(head,substitutewords_strings[i][1],substitutewords_strings[i][2])
746   end
747   return head
748 end

```

10.19 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.

```

749 tabularasa_onlytext = false
750
751 tabularasa = function(head)
752   local s = nodenew(nodeid"kern")
753   for line in nodetraverseid(nodeid"hlist",head) do
754     for n in nodetraverseid(nodeid"glyph",line.head) do
755       if not(tabularasa_onlytext) or node.has_attribute(n,luatexbase.attributes.tabularasaattr) then
756         s.kern = n.width
757         nodeinsertafter(line.list,n,nodecopy(s))
758         line.head = noderemove(line.list,n)
759       end
760     end
761   end
762   return head
763 end

```

10.20 uppercasecolor

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

```

764 uppercasecolor_onlytext = false
765
766 uppercasecolor = function (head)
767   for line in nodetraverseid(Hhead,head) do
768     for upper in nodetraverseid(GLYPH,line.head) do

```



```

769     if not(uppercasecolor_onlytext) or node.has_attribute(upper,luatexbase.attributes.uppercase
770         if (((upper.char > 64) and (upper.char < 91)) or
771             ((upper.char > 57424) and (upper.char < 57451))) then -- for small caps! nice
772             color_push.data = randomcolorstring() -- color or grey string
773             line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
774             nodeinsertafter(line.head,upper,nodecopy(color_pop))
775         end
776     end
777 end
778 end
779 return head
780 end

```

10.21 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 \TeX . 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!

10.21.1 colorstretch – preliminaries

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

```

781 keeptext = true
782 colorexpansion = true
783
784 colorstretch_coloroffset = 0.5
785 colorstretch_colorange = 0.5
786 chickenize_rule_bad_height = 4/5 -- height and depth of the rules
787 chickenize_rule_bad_depth = 1/5
788
789
790 colorstretchnumbers = true
791 drawstretchthreshold = 0.1
792 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.

```

793 colorstretch = function (head)
794   local f = font.getfont(font.current()).characters
795   for line in nodetraverseid(Hhead,head) do
796     local rule_bad = nodenew(RULE)
797
798     if colorexpanansion then -- if also the font expansion should be shown
799       local g = line.head
800       while not(g.id == 37) do
801         g = g.next
802       end
803       exp_factor = g.width / f[g.char].width
804       exp_color = colorstretch_coloroffset + (1-exp_factor)*10 .. " g"
805       rule_bad.width = 0.5*line.width -- we need two rules on each line!
806     else
807       rule_bad.width = line.width -- only the space expansion should be shown, only one rule
808     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.

```

809   rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bet
810   rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth
811
812   local glue_ratio = 0
813   if line.glue_order == 0 then
814     if line.glue_sign == 1 then
815       glue_ratio = colorstretch_colorrage * math.min(line.glue_set,1)
816     else
817       glue_ratio = -colorstretch_colorrage * math.min(line.glue_set,1)
818     end
819   end
820   color_push.data = colorstretch_coloroffset + glue_ratio .. " g"
821

```

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

```

822 -- set up output
823   local p = line.head
824
825   -- a rule to immitate kerning all the way back
826   local kern_back = nodenew(RULE)
827   kern_back.width = -line.width
828
829   -- if the text should still be displayed, the color and box nodes are inserted additionally
830   -- and the head is set to the color node
831   if keptext then
832     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
833   else
834     node.flush_list(p)

```

```

835     line.head = nodecopy(color_push)
836 end
837 nodeinsertafter(line.head,line.head,rule_bad) -- then the rule
838 nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
839 tmpnode = nodeinsertafter(line.head,line.head.next.next,kern_back)
840
841 -- then a rule with the expansion color
842 if colorexansion then -- if also the stretch/shrink of letters should be shown
843     color_push.data = exp_color
844     nodeinsertafter(line.head,tmpnode,nodecopy(color_push))
845     nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
846     nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
847 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.

```

848 if colorstretchnumbers then
849     j = 1
850     glue_ratio_output = {}
851     for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
852         local char = unicode.utf8.char(s)
853         glue_ratio_output[j] = nodenew(37,1)
854         glue_ratio_output[j].font = font.current()
855         glue_ratio_output[j].char = s
856         j = j+1
857     end
858     if math.abs(glue_ratio) > drawstretchthreshold then
859         if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
860         else color_push.data = "0 0.99 0 rg" end
861     else color_push.data = "0 0 0 rg"
862     end
863
864     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
865     for i = 1,math.min(j-1,7) do
866         nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])
867     end
868     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
869 end -- end of stretch number insertion
870 end
871 return head
872 end

```

dubstepize

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

873

scorpionize

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

```
874 function scorpionize_color(head)
875   color_push.data = ".35 .55 .75 rg"
876   nodeinsertafter(head,head,nodecopy(color_push))
877   nodeinsertafter(head,node.tail(head),nodecopy(color_pop))
878   return head
879 end
```

10.22 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`.

10.22.1 zebranize – preliminaries

```
880 zebracolorarray = {}
881 zebracolorarray_bg = {}
882 zebracolorarray[1] = "0.1 g"
883 zebracolorarray[2] = "0.9 g"
884 zebracolorarray_bg[1] = "0.9 g"
885 zebracolorarray_bg[2] = "0.1 g"
```

10.22.2 zebranize – the function

This code has to be revisited, it is ugly.

```
886 function zebranize(head)
887   zebracolor = 1
888   for line in nodetraverseid(nodeid"hhead",head) do
889     if zebracolor == #zebracolorarray then zebracolor = 0 end
890     zebracolor = zebracolor + 1
891     color_push.data = zebracolorarray[zebracolor]
892     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
893     for n in nodetraverseid(nodeid"glyph",line.head) do
894       if n.next then else
```

```

895         nodeinsertafter(line.head,n,nodecopy(color_pull))
896     end
897 end
898
899 local rule_zebra = nodenew(RULE)
900 rule_zebra.width = line.width
901 rule_zebra.height = tex.baselineskip.width*4/5
902 rule_zebra.depth = tex.baselineskip.width*1/5
903
904 local kern_back = nodenew(RULE)
905 kern_back.width = -line.width
906
907 color_push.data = zebracolorarray_bg[zebracolor]
908 line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
909 line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
910 nodeinsertafter(line.head,line.head,kern_back)
911 nodeinsertafter(line.head,line.head,rule_zebra)
912 end
913 return (head)
914 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!

```

915 --
916 function pdf_print (...)
917   for _, str in ipairs({...}) do
918     pdf.print(str .. " ")
919   end
920   pdf.print("\string\n")
921 end
922
923 function move (p)
924   pdf_print(p[1],p[2],"m")
925 end
926
927 function line (p)
928   pdf_print(p[1],p[2],"l")
929 end
930
931 function curve(p1,p2,p3)
932   pdf_print(p1[1], p1[2],
933             p2[1], p2[2],
934             p3[1], p3[2], "c")
935 end
936
937 function close ()
938   pdf_print("h")
939 end
940
941 function linewidth (w)
942   pdf_print(w,"w")
943 end
944
945 function stroke ()
946   pdf_print("S")
947 end
948 --
949

```

```

950 function strictcircle(center,radius)
951   local left = {center[1] - radius, center[2]}
952   local lefttop = {left[1], left[2] + 1.45*radius}
953   local leftbot = {left[1], left[2] - 1.45*radius}
954   local right = {center[1] + radius, center[2]}
955   local righttop = {right[1], right[2] + 1.45*radius}
956   local rightbot = {right[1], right[2] - 1.45*radius}
957
958   move (left)
959   curve (lefttop, righttop, right)
960   curve (rightbot, leftbot, left)
961 stroke()
962 end
963
964 function disturb_point(point)
965   return {point[1] + math.random()*5 - 2.5,
966           point[2] + math.random()*5 - 2.5}
967 end
968
969 function sloppycircle(center,radius)
970   local left = disturb_point({center[1] - radius, center[2]})
971   local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
972   local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}
973   local right = disturb_point({center[1] + radius, center[2]})
974   local righttop = disturb_point({right[1], right[2] + 1.45*radius})
975   local rightbot = disturb_point({right[1], right[2] - 1.45*radius})
976
977   local right_end = disturb_point(right)
978
979   move (right)
980   curve (rightbot, leftbot, left)
981   curve (lefttop, righttop, right_end)
982   linewidth(math.random()+0.5)
983   stroke()
984 end
985
986 function sloppyline(start,stop)
987   local start_line = disturb_point(start)
988   local stop_line = disturb_point(stop)
989   start = disturb_point(start)
990   stop = disturb_point(stop)
991   move(start) curve(start_line,stop_line,stop)
992   linewidth(math.random()+0.5)
993   stroke()
994 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 ...