

*»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 (chosen by the user)
variantjustification	Justification by using glyph variants

less useful functions

boustrophedon	invert every second line in the style of archaic greek texts
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	7
2.3	\text-Versions	7
2.4	Lua functions	7
3	Options – How to Adjust It	8
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	27
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	variantjustification	39
10.24	zebranize	40
10.24.1	zebranize – preliminaries	40
10.24.2	zebranize – the function	40
11	Drawing	42
12	Known Bugs	44
13	To Do's	44
14	Literature	44
15	Thanks	44

Part I

User Documentation

1 How It Works

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

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

2 Commands – How You Can Use It

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

2.1 _T_E_X Commands – Document Wide

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

The effect of the commands can be influenced, not with arguments, but only via the `\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!

\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.

\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!

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

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

\nyanize A synonym for `rainbowcolor`.

\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.

\randomuclc 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.

\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 ...

\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.

\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.

\variantjustification For special document types, it might be mandatory to have a fixed interword space. If you still want to have a justified type area, there must be another kind of stretchable material – one version realized by this command is using wide variants of glyphs to fill the remaining space. As the glyph substitution takes place randomly, this does *not* provide the optimum justification, as this would take up much computation power.

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.

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`!

3 Options – How to Adjust It

There are several ways to change the behaviour of `chickenize` and its macros. Most of the options are Lua variables and can be set using `\chickenizesetup`. But be *careful!* The argument of `\chickenizesetup` is passed directly to Lua, therefore you are *not* using a comma-separated key-value list, but uncorrelated Lua commands. The argument must have the syntax `{randomfontslower = 1 randomfontsupper = 0}` instead of `{randomfontslower = 1, randomfontsupper = 0}`. Alright?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`colorexpansion = <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 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\boustrophedonglyphs{
33   \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon_glyphs,"boustrophedon_glyphs")}
34 \def\unboustrophedonglyphs{
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",gutenbergize_rq,"gutenbergize_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",pancakertext,"pancakertext")}}
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\variantjustification{
238 \directlua{luatexbase.add_to_callback("post_linebreak_filter",variantjustification,"variantjust.
239 \def\unvariantjustification{
240 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","variantjustification")}}
241
242 \def\zebranize{
243 \directlua{luatexbase.add_to_callback("post_linebreak_filter",zebranize,"zebranize")}}
244 \def\unzebranize{
245 \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.

```

246 \newluatexattribute\leetattr
247 \newluatexattribute\letterspaceadjustattr
248 \newluatexattribute\randcolorattr
249 \newluatexattribute\randfontsattrib
250 \newluatexattribute\randucattr
251 \newluatexattribute\tabularasaattr
252 \newluatexattribute\uppercasecolorattr
253
254 \long\def\textleetspeak#1%
255 {\setluatexattribute\leetattr{42}#1\unsetluatexattribute\leetattr}
256
257 \long\def\textletterspaceadjust#1{
258 \setluatexattribute\letterspaceadjustattr{42}#1\unsetluatexattribute\letterspaceadjustattr
259 \directlua{

```

```

260     if (textletterspaceadjustactive) then else % -- if already active, do nothing
261         luatexbase.add_to_callback("pre_linebreak_filter",textletterspaceadjust,"textletterspaceadj
262     end
263     textletterspaceadjustactive = true           % -- set to active
264 }
265 }
266 \let\textlsa\textletterspaceadjust
267
268 \long\def\textrandomcolor#1%
269     {\setluatexattribute\randcolorattr{42}#1\unsetluatexattribute\randcolorattr}
270 \long\def\textrandomfonts#1%
271     {\setluatexattribute\randfontssattr{42}#1\unsetluatexattribute\randfontssattr}
272 \long\def\textrandomfontsf#1%
273     {\setluatexattribute\randfontssattr{42}#1\unsetluatexattribute\randfontssattr}
274 \long\def\textrandomuclc#1%
275     {\setluatexattribute\randuclcatr{42}#1\unsetluatexattribute\randuclcatr}
276 \long\def\texttabularasa#1%
277     {\setluatexattribute\tabularasaattr{42}#1\unsetluatexattribute\tabularasaattr}
278 \long\def\textuppercasecolor#1%
279     {\setluatexattribute\uppercasecolorattr{42}#1\unsetluatexattribute\uppercasecolorattr}

```

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

```

280 \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.

```

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

```

```

301
302 bein21 = {270,-50}
303 bein22 = {260,-75}
304 bein23 = {245,-75}
305
306 schnabel_oben = {185,55}
307 schnabel_vorne = {165,45}
308 schnabel_unten = {185,35}
309
310 flugel_vorne = {260,-10}
311 flugel_unten = {280,-40}
312 flugel_hinten = {275,-15}
313
314 sloppycircle(kopf,kopf_rad)
315 sloppyline(d,e)
316 sloppycircle(korper,korper_rad)
317 sloppyline(bein11,bein12) sloppyline(bein12,bein13)
318 sloppyline(bein21,bein22) sloppyline(bein22,bein23)
319 sloppyline(schnabel_vorne,schnabel_oben) sloppyline(schnabel_vorne,schnabel_unten)
320 sloppyline(flugel_vorne,flugel_unten) sloppyline(flugel_hinten,flugel_unten)
321 }
322 }

```

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.

```

323 \ProvidesPackage{chickenize}%
324 [2012/09/16 v0.1a chickenize package]
325 \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! :)

```

326 \iffalse
327 \DeclareDocumentCommand\includegraphics{0}{m}{
328   \fbox{Chicken}  %% actually, I'd love to draw an MP graph showing a chicken ...
329 }
330 %%% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
331 %% So far, you have to load pgfplots yourself.

```

```

332 %% As it is a mighty package, I don't want the user to force loading it.
333 \NewDocumentCommand\balmerpeak{G{}0{-4cm}}{
334 %% to be done using Lua drawing.
335 }
336 \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.

```

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

```

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

```

353 color_push = nodenew(WHAT,COL)
354 color_pop = nodenew(WHAT,COL)
355 color_push.stack = 0
356 color_pop.stack = 0
357 color_push.cmd = 1
358 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.

```

359 chicken_pagenumbers = true
360
361 chickenstring = {}
362 chickenstring[1] = "chicken" -- chickenstring is a table, please remeber this!

```

```

363
364 chickenizefraction = 0.5
365 -- set this to a small value to fool somebody, or to see if your text has been read carefully. Th
366 chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing your
367
368 local tbl = font.getfont(font.current())
369 local space = tbl.parameters.space
370 local shrink = tbl.parameters.space_shrink
371 local stretch = tbl.parameters.space_stretch
372 local match = unicode.utf8.match
373 chickenize_ignore_word = false

```

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

```

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

```

```

408     chicken[1] = node.kerning(chicken[1])    -- FIXME: does not work
409     chicken[1] = node.ligaturing(chicken[1]) -- dito
410
411     nodeinsertbefore(head,i,chicken[1])
412     chicken[1].next = chicken[2] -- seems to be necessary ... to be fixed
413     chicken[string.len(chickenstring_tmp)].next = i.next
414
415     -- shift lowercase latin letter to uppercase if the original input was an uppercase
416     if (chickenize_capital and (chicken[1].char > 96 and chicken[1].char < 123)) then
417         chicken[1].char = chicken[1].char - 32
418     end
419
420     return head
421 end
422
423 chickenize = function(head)
424     for i in nodetraverseid(37,head) do --find start of a word
425         if (chickenize_ignore_word == false) then -- normal case: at the beginning of a word, we jump
426             if (i.char > 64 and i.char < 91) then chickenize_capital = true else chickenize_capital = false
427             head = chickenize_real_stuff(i,head)
428         end
429
430 -- At the end of the word, the ignoring is reset. New chance for everyone.
431         if not((i.next.id == 37) or (i.next.id == 7) or (i.next.id == 22) or (i.next.id == 11)) then
432             chickenize_ignore_word = false
433         end
434
435 -- And the random determination of the chickenization of the next word:
436         if math.random() > chickenizefraction then
437             chickenize_ignore_word = true
438         elseif chickencount then
439             chicken_substitutions = chicken_substitutions + 1
440         end
441     end
442     return head
443 end
444

```

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)

```

445 local separator      = string.rep("=", 28)
446 local texiowrite_nl = texio.write_nl
447 nicetext = function()
448     texiowrite_nl("Output written on "..tex.jobname.." pdf ("..status.total_pages.." chicken,".." eg
449     texiowrite_nl(" ")
450     texiowrite_nl(separator)
451     texiowrite_nl("Hello my dear user,")

```

```

452 texiowrite_nl("good job, now go outside and enjoy the world!")
453 texiowrite_nl(" ")
454 texiowrite_nl("And don't forget to feed your chicken!")
455 texiowrite_nl(separator .. "\n")
456 if chickencount then
457     texiowrite_nl("There were "..chicken_substitutions.." substitutions made.")
458     texiowrite_nl(separator)
459 end
460 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:

```

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

```

Glyphwise rotation:

```

479 boustrophedon_glyphs = function(head)
480     odd = false
481     rot = nodenew(8,8)
482     rot2 = nodenew(8,8)
483     for line in nodetraverseid(0,head) do
484         if odd==true then
485             line.dir = "TRT"
486             for g in nodetraverseid(37,line.head) do
487                 w = -g.width/65536*0.99625
488                 rot.data = "-1 0 0 1 " .. w .." 0 cm"
489                 rot2.data = "-1 0 0 1 " .. -w .." 0 cm"

```



```

490     line.head = node.insert_before(line.head,g,node.copy(rot))
491     node.insert_after(line.head,g,node.copy(rot2))
492 end
493 odd = false
494 else
495     line.dir = "TLT"
496     odd = true
497 end
498 end
499 return head
500 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.

```

501 boustrophedon_inverse = function(head)
502   rot = node.new(8,8)
503   rot2 = node.new(8,8)
504   odd = true
505   for line in node.traverse_id(0,head) do
506     if odd == false then
507 texio.write_nl(line.height)
508     w = line.width/65536*0.99625 -- empirical correction factor (?)
509     h = line.height/65536*0.99625
510     rot.data = "-1 0 0 -1 "..w.." "..h.." cm"
511     rot2.data = "-1 0 0 -1"..-w.." "..0.5*h.." cm"
512     line.head = node.insert_before(line.head,line.head,node.copy(rot))
513     node.insert_after(line.head,node.tail(line.head),node.copy(rot2))
514     odd = true
515   else
516     odd = false
517   end
518 end
519 return head
520 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.

```

521 countglyphs = function(head)
522   for line in nodetraverseid(0,head) do
523     for glyph in nodetraverseid(37,line.head) do
524       glyphnumber = glyphnumber + 1

```

```

525     end
526 end
527 return head
528 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? ...

```

529 printglyphnumber = function()
530   texiowrite_nl("Number of glyphs in this document: "..glyphnumber.."\\n")
531 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 `\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.

10.4.1 guttenbergenize – preliminaries

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

```

532 local quotestrings = {
533   [171] = true, [172] = true,
534   [8216] = true, [8217] = true, [8218] = true,
535   [8219] = true, [8220] = true, [8221] = true,
536   [8222] = true, [8223] = true,
537   [8248] = true, [8249] = true, [8250] = true,
538 }

```

10.4.2 guttenbergenize – the function

```

539 guttenbergenize_rq = function(head)
540   for n in nodetraverseid(nodeid"glyph",head) do
541     local i = n.char
542     if quotestrings[i] then
543       noderemove(head,n)
544     end
545   end
546   return head
547 end

```

⁹Thanks to Jasper for bringing me to this idea!

10.5 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.¹⁰

```
548 hammertimedelay = 1.2
549 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
550 hammertime = function(head)
551   if hammerfirst then
552     texiowrite_nl(htime_separator)
553     texiowrite_nl("=====STOP!=====\\n")
554     texiowrite_nl(htime_separator .. "\\n\\n\\n")
555     os.sleep (hammertimedelay*1.5)
556     texiowrite_nl(htime_separator .. "\\n")
557     texiowrite_nl("=====HAMMERTIME=====\\n")
558     texiowrite_nl(htime_separator .. "\\n\\n")
559     os.sleep (hammertimedelay)
560     hammerfirst = false
561   else
562     os.sleep (hammertimedelay)
563     texiowrite_nl(htime_separator)
564     texiowrite_nl("=====U can't touch this!=====\\n")
565     texiowrite_nl(htime_separator .. "\\n\\n")
566     os.sleep (hammertimedelay*0.5)
567   end
568   return head
569 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.

```
570 itsame = function()
571 local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
572 color = "1 .6 0"
573 for i = 6,9 do mr(i,3) end
574 for i = 3,11 do mr(i,4) end
575 for i = 3,12 do mr(i,5) end
576 for i = 4,8 do mr(i,6) end
577 for i = 4,10 do mr(i,7) end
578 for i = 1,12 do mr(i,11) end
579 for i = 1,12 do mr(i,12) end
580 for i = 1,12 do mr(i,13) end
581
```

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

```

582 color = ".3 .5 .2"
583 for i = 3,5 do mr(i,3) end mr(8,3)
584 mr(2,4) mr(4,4) mr(8,4)
585 mr(2,5) mr(4,5) mr(5,5) mr(9,5)
586 mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
587 for i = 3,8 do mr(i,8) end
588 for i = 2,11 do mr(i,9) end
589 for i = 1,12 do mr(i,10) end
590 mr(3,11) mr(10,11)
591 for i = 2,4 do mr(i,15) end for i = 9,11 do mr(i,15) end
592 for i = 1,4 do mr(i,16) end for i = 9,12 do mr(i,16) end
593
594 color = "1 0 0"
595 for i = 4,9 do mr(i,1) end
596 for i = 3,12 do mr(i,2) end
597 for i = 8,10 do mr(5,i) end
598 for i = 5,8 do mr(i,10) end
599 mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
600 for i = 4,9 do mr(i,12) end
601 for i = 3,10 do mr(i,13) end
602 for i = 3,5 do mr(i,14) end
603 for i = 7,10 do mr(i,14) end
604 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.

```

605 chickenkernamount = 0
606 chickeninvertkerning = false
607
608 function kernmanipulate (head)
609   if chickeninvertkerning then -- invert the kerning
610     for n in nodetraverseid(11,head) do
611       n.kern = -n.kern
612     end
613   else -- if not, set it to the given value
614     for n in nodetraverseid(11,head) do
615       n.kern = chickenkernamount
616     end
617   end
618   return head
619 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.

```
620 leetspeak_onlytext = false
621 leettable = {
622   [101] = 51, -- E
623   [105] = 49, -- I
624   [108] = 49, -- L
625   [111] = 48, -- O
626   [115] = 53, -- S
627   [116] = 55, -- T
628
629   [101-32] = 51, -- e
630   [105-32] = 49, -- i
631   [108-32] = 49, -- l
632   [111-32] = 48, -- o
633   [115-32] = 53, -- s
634   [116-32] = 55, -- t
635 }
```

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

```
636 leet = function(head)
637   for line in nodetraverseid(Hhead,head) do
638     for i in nodetraverseid(GLYPH,line.head) do
639       if not leetspeak_onlytext or
640         node.has_attribute(i,luatexbase.attributes.leetattr)
641       then
642         if leettable[i.char] then
643           i.char = leettable[i.char]
644         end
645       end
646     end
647   end
648   return head
649 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

```
650 local letterspace_glue = nodenew(nodeid"glue")
651 local letterspace_spec = nodenew(nodeid"glue_spec")
652 local letterspace_pen = nodenew(nodeid"penalty")
653
654 letterspace_spec.width = tex.sp"0pt"
655 letterspace_spec.stretch = tex.sp"2pt"
656 letterspace_glue.spec = letterspace_spec
657 letterspace_pen.penalty = 10000
```

10.9.2 function implementation

```
658 letterspaceadjust = function(head)
659   for glyph in nodetraverseid(nodeid"glyph", head) do
660     if glyph.prev and (glyph.prev.id == nodeid"glyph" or glyph.prev.id == nodeid"disc") then
661       local g = nodecopy(letterspace_glue)
662       nodeinsertbefore(head, glyph, g)
663       nodeinsertbefore(head, g, nodecopy(letterspace_pen))
664     end
665   end
666   return head
667 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!

```
668 textletterspaceadjust = function(head)
669   for glyph in node.traverse_id(node.id"glyph", head) do
670     if node.has_attribute(glyph, luatexbase.attributes.letterspaceadjustattr) then
671       if glyph.prev and (glyph.prev.id == node.id"glyph" or glyph.prev.id == node.id"disc") then
672         local g = node.copy(letterspace_glue)
673         node.insert_before(head, glyph, g)
674         node.insert_before(head, g, node.copy(letterspace_pen))
675       end
676     end
677   end
678   luatexbase.remove_from_callback("pre_linebreak_filter", "textletterspaceadjust")
679   return head
680 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.

```

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

```

10.11 pancakenize

```

702 local separator      = string.rep("=", 28)
703 local texiowrite_nl = texio.write_nl
704 pancaketext = function()
705   texiowrite_nl("Output written on "..tex.jobname.." pdf ("..status.total_pages.." chicken, ".." eg
706   texiowrite_nl(" ")
707   texiowrite_nl(separator)
708   texiowrite_nl("Soo ... you decided to use \\pancakenize.")
709   texiowrite_nl("That means you owe me a pancake!")
710   texiowrite_nl(" ")
711   texiowrite_nl("(This goes by document, not compilation.)")
712   texiowrite_nl(separator.."\\n\\n")
713   texiowrite_nl("Looking forward for my pancake! :)")
714   texiowrite_nl("\\n\\n")
715 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.

```

716 randomfontslower = 1
717 randomfontsupper = 0

```

```

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

```

10.14 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```

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

```

10.15 randomchars

```

748 randomchars = function(head)
749   for line in nodetraverseid(Hhead,head) do
750     for i in nodetraverseid(GLYPH,line.head) do
751       i.char = math.floor(math.random()*512)
752     end
753   end
754   return head
755 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.

```
756 randomcolor_grey = false
757 randomcolor_onlytext = false --switch between local and global colorization
758 rainbowcolor = false
759
760 grey_lower = 0
761 grey_upper = 900
762
763 Rgb_lower = 1
764 rGb_lower = 1
765 rgB_lower = 1
766 Rgb_upper = 254
767 rGb_upper = 254
768 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.

```
769 rainbow_step = 0.005
770 rainbow_Rgb = 1-rainbow_step -- we start in the red phase
771 rainbow_rGb = rainbow_step -- values x must always be  $0 < x < 1$ 
772 rainbow_rgB = rainbow_step
773 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.

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

```

793     if rainbow_rgb <= rainbow_step then rainind = 1 end
794     end
795     return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rgB.." rg"
796 else
797     Rgb = math.random(Rgb_lower,Rgb_upper)/255
798     rGb = math.random(rGb_lower,rGb_upper)/255
799     rgB = math.random(rgB_lower,rgB_upper)/255
800     return Rgb.." "..rGb.." "..rgB.." .." rg"
801 end
802 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.

```

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

```

10.17 randomerror

```

817 %

```

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.

```

818 substitutewords_strings = {}
819
820 addtosubstitutions = function(input,output)
821   substitutewords_strings[#substitutewords_strings + 1] = {}
822   substitutewords_strings[#substitutewords_strings][1] = input
823   substitutewords_strings[#substitutewords_strings][2] = output
824 end
825
826 substitutewords = function(head)
827   for i = 1,#substitutewords_strings do
828     head = string.gsub(head,substitutewords_strings[i][1],substitutewords_strings[i][2])
829   end
830   return head
831 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.

```

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

```

10.21 uppercasecolor

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

```

847 uppercasecolor_onlytext = false
848
849 uppercasecolor = function (head)
850   for line in nodetraverseid(Hhead,head) do
851     for upper in nodetraverseid(GLYPH,line.head) do

```

```

852     if not(uppercasecolor_onlytext) or node.has_attribute(upper,luatexbase.attributes.uppercase
853         if (((upper.char > 64) and (upper.char < 91)) or
854             ((upper.char > 57424) and (upper.char < 57451))) then -- for small caps! nice
855             color_push.data = randomcolorstring() -- color or grey string
856             line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
857             nodeinsertafter(line.head,upper,nodecopy(color_pop))
858         end
859     end
860 end
861 end
862 return head
863 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.

```

864 keeptext = true
865 colorexpansion = true
866
867 colorstretch_coloroffset = 0.5
868 colorstretch_colorange = 0.5
869 chickenize_rule_bad_height = 4/5 -- height and depth of the rules
870 chickenize_rule_bad_depth = 1/5
871
872
873 colorstretchnumbers = true
874 drawstretchthreshold = 0.1
875 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.

```

876 colorstretch = function (head)
877   local f = font.getfont(font.current()).characters
878   for line in nodetraverseid(Hhead,head) do
879     local rule_bad = nodenew(RULE)
880
881     if colorexpanansion then -- if also the font expansion should be shown
882       local g = line.head
883       while not(g.id == 37) and (g.next) do g = g.next end -- find first glyph on line. If line is
884       if (g.id == 37) then -- read width only if g is a glyph!
885         exp_factor = g.width / f[g.char].width
886         exp_color = colorstretch_coloroffset + (1-exp_factor)*10 .. " g"
887         rule_bad.width = 0.5*line.width -- we need two rules on each line!
888       end
889     else
890       rule_bad.width = line.width -- only the space expansion should be shown, only one rule
891     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.

```

892   rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bet
893   rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth
894
895   local glue_ratio = 0
896   if line.glue_order == 0 then
897     if line.glue_sign == 1 then
898       glue_ratio = colorstretch_colorrage * math.min(line.glue_set,1)
899     else
900       glue_ratio = -colorstretch_colorrage * math.min(line.glue_set,1)
901     end
902   end
903   color_push.data = colorstretch_coloroffset + glue_ratio .. " g"
904

```

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

```

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

```

```

918     line.head = nodecopy(color_push)
919 end
920 nodeinsertafter(line.head,line.head,rule_bad) -- then the rule
921 nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
922 tmpnode = nodeinsertafter(line.head,line.head.next.next,kern_back)
923
924 -- then a rule with the expansion color
925 if colorexansion then -- if also the stretch/shrink of letters should be shown
926     color_push.data = exp_color
927     nodeinsertafter(line.head,tmpnode,nodecopy(color_push))
928     nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
929     nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
930 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.

```

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

```

dubstepize

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

956

scorpionize

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

```
957 function scorpionize_color(head)
958   color_push.data = ".35 .55 .75 rg"
959   nodeinsertafter(head,head,nodecopy(color_push))
960   nodeinsertafter(head,node.tail(head),nodecopy(color_pop))
961   return head
962 end
```

10.23 variantjustification

The list `substlist` defines which glyphs can be replaced by others. Use the unicode code points for this. So far, only wider variants are possible! Extend the list at will. If you find useful definitions, send me any glyph combination!

Some predefined values for hebrew typesetting; the list is not local so the user can change it in a very transparent way (using `\chickenizesetup{}`). This costs runtime, however ... I guess ... (?)

```
963 substlist = {}
964 substlist[1488] = 64289
965 substlist[1491] = 64290
966 substlist[1492] = 64291
967 substlist[1499] = 64292
968 substlist[1500] = 64293
969 substlist[1501] = 64294
970 substlist[1512] = 64295
971 substlist[1514] = 64296
```

In the function, we need reproduceable randomization so every compilation of the same document looks the same. Else this would make contracts invalid.

The last line is excluded from the procedure as it makes no sense to extend it this way. If you really want to typeset a rectangle, use the appropriate way to disable the space at the end of the paragraph (german “Ausgang”).

```
972 function variantjustification(head)
973   math.randomseed(1)
974   for line in nodetraverseid(nodeid"hhead",head) do
975     if (line.glue_sign == 1 and line.glue_order == 0) then -- exclude the last line!
976       substitutions_wide = {} -- we store all "expandable" letters of each line
977       for n in nodetraverseid(nodeid"glyph",line.head) do
978         if (substlist[n.char]) then
979           substitutions_wide[#substitutions_wide+1] = n
980         end
981       end
982     end
983   end
```

```

981     end
982     line.glue_set = 0    -- deactivate normal glue expansion
983     local width = node.dimensions(line.head)  -- check the new width of the line
984     local goal = line.width
985     while (width < goal and #substitutions_wide > 0) do
986         x = math.random(#substitutions_wide)    -- choose randomly a glyph to be substituted
987         oldchar = substitutions_wide[x].char
988         substitutions_wide[x].char = substlist[substitutions_wide[x].char] -- substitute by wide
989         width = node.dimensions(line.head)      -- check if the line is too wide
990         if width > goal then substitutions_wide[x].char = oldchar break end -- substitute back if
991         table.remove(substitutions_wide,x)      -- if further substitutions have to be done,
992     end
993 end
994 end
995 return head
996 end

```

That's it. Actually, the function is quite simple and should work out of the box. However, small columns will most probably not work as there typically is not much expandable stuff in a normal line of text.

10.24 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`.

10.24.1 zebranize – preliminaries

```

997 zebracolorarray = {}
998 zebracolorarray_bg = {}
999 zebracolorarray[1] = "0.1 g"
1000 zebracolorarray[2] = "0.9 g"
1001 zebracolorarray_bg[1] = "0.9 g"
1002 zebracolorarray_bg[2] = "0.1 g"

```

10.24.2 zebranize – the function

This code has to be revisited, it is ugly.

```

1003 function zebranize(head)
1004     zebracolor = 1
1005     for line in nodetraverseid(nodeid"hhead",head) do
1006         if zebracolor == #zebracolorarray then zebracolor = 0 end
1007         zebracolor = zebracolor + 1
1008         color_push.data = zebracolorarray[zebracolor]

```



```

1009     line.head =      nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1010     for n in nodetraverseid(nodeid"glyph",line.head) do
1011         if n.next then else
1012             nodeinsertafter(line.head,n,nodecopy(color_pull))
1013         end
1014     end
1015
1016     local rule_zebra = nodenew(RULE)
1017     rule_zebra.width = line.width
1018     rule_zebra.height = tex.baselineskip.width*4/5
1019     rule_zebra.depth = tex.baselineskip.width*1/5
1020
1021     local kern_back = nodenew(RULE)
1022     kern_back.width = -line.width
1023
1024     color_push.data = zebracolorarray_bg[zebracolor]
1025     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
1026     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1027     nodeinsertafter(line.head,line.head,kern_back)
1028     nodeinsertafter(line.head,line.head,rule_zebra)
1029 end
1030 return (head)
1031 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!

```
1032 --
1033 function pdf_print (...)
1034   for _, str in ipairs({...}) do
1035     pdf.print(str .. " ")
1036   end
1037   pdf.print("\string\n")
1038 end
1039
1040 function move (p)
1041   pdf_print(p[1],p[2],"m")
1042 end
1043
1044 function line (p)
1045   pdf_print(p[1],p[2],"l")
1046 end
1047
1048 function curve(p1,p2,p3)
1049   pdf_print(p1[1], p1[2],
1050             p2[1], p2[2],
1051             p3[1], p3[2], "c")
1052 end
1053
1054 function close ()
1055   pdf_print("h")
1056 end
1057
1058 function linewidth (w)
1059   pdf_print(w,"w")
1060 end
1061
1062 function stroke ()
1063   pdf_print("S")
1064 end
1065 --
1066
```

```

1067 function strictcircle(center,radius)
1068   local left = {center[1] - radius, center[2]}
1069   local lefttop = {left[1], left[2] + 1.45*radius}
1070   local leftbot = {left[1], left[2] - 1.45*radius}
1071   local right = {center[1] + radius, center[2]}
1072   local righttop = {right[1], right[2] + 1.45*radius}
1073   local rightbot = {right[1], right[2] - 1.45*radius}
1074
1075   move (left)
1076   curve (lefttop, righttop, right)
1077   curve (rightbot, leftbot, left)
1078   stroke()
1079 end
1080
1081 function disturb_point(point)
1082   return {point[1] + math.random()*5 - 2.5,
1083           point[2] + math.random()*5 - 2.5}
1084 end
1085
1086 function sloppycircle(center,radius)
1087   local left = disturb_point({center[1] - radius, center[2]})
1088   local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
1089   local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}
1090   local right = disturb_point({center[1] + radius, center[2]})
1091   local righttop = disturb_point({right[1], right[2] + 1.45*radius})
1092   local rightbot = disturb_point({right[1], right[2] - 1.45*radius})
1093
1094   local right_end = disturb_point(right)
1095
1096   move (right)
1097   curve (rightbot, leftbot, left)
1098   curve (lefttop, righttop, right_end)
1099   linewidth(math.random()+0.5)
1100   stroke()
1101 end
1102
1103 function sloppyline(start,stop)
1104   local start_line = disturb_point(start)
1105   local stop_line = disturb_point(stop)
1106   start = disturb_point(start)
1107   stop = disturb_point(stop)
1108   move(start) curve(start_line,stop_line,stop)
1109   linewidth(math.random()+0.5)
1110   stroke()
1111 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 ...