



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

Paul Isambert

CHICKENIZE

v0.2.2

Arno L. Trautmann \AA
arno.trautmann@gmx.de

[How to read this document.](#)

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, which might never happen.

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 © 2015 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 line-wise
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
suppressonecharbreak	suppresses linebreaks after single-letter words

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
countwords	counts the number of words in the whole document
leetspeak	translates the (latin-based) input into 1337 5p34k
medievalumlaut	changes each umlaut to normal glyph plus “e” above it: âôû
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	8
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	22
9.1	Free Compliments	22
9.2	Definition of User-Level Macros	22
10	Lua Module	23
10.1	chickenize	23
10.2	boustrophedon	26
10.3	bubblesort	27
10.4	countglyphs	27
10.5	countwords	28
10.6	detectdoublewords	29
10.7	gutenbergize	29
10.7.1	gutenbergize – preliminaries	30
10.7.2	gutenbergize – the function	30
10.8	hammertime	30
10.9	itsame	31

10.10 kernmanipulate	32
10.11 leetspeak	32
10.12 leftsideright	33
10.13 letterspaceadjust	33
10.13.1 setup of variables	33
10.13.2 function implementation	34
10.13.3 textletterspaceadjust	34
10.14 matrixize	34
10.15 medievalumlaut	35
10.16 pancakenize	36
10.17 randomerror	36
10.18 randomfonts	36
10.19 randomucl	37
10.20 randomchars	37
10.21 randomcolor and rainbowcolor	38
10.21.1 randomcolor – preliminaries	38
10.21.2 randomcolor – the function	39
10.22 randomerror	39
10.23 rickroll	39
10.24 substitutewords	39
10.25 suppressonecharbreak	40
10.26 tabularasa	40
10.27 tanjanize	41
10.28 uppercasecolor	42
10.29 upsidedown	42
10.30 colorstretch	43
10.30.1 colorstretch – preliminaries	43
10.31 variantjustification	46
10.32 zebranize	47
10.32.1 zebranize – preliminaries	47
10.32.2 zebranize – the function	47
11 Drawing	49
12 Known Bugs	51
13 To Do’s	51
14 Literature	51
15 Thanks	51

Part I

User Documentation

1 How It Works

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

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

2 Commands – How You Can Use It

There are several ways to make use of the *chickenize* package – you can either stay on the $\text{T}_{\text{E}}\text{X}$ side or use the Lua functions directly. In fact, the $\text{T}_{\text{E}}\text{X}$ macros are simple wrappers around the functions.

2.1 $\text{T}_{\text{E}}\text{X}$ Commands – Document Wide

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

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

`\allownumberincommands` Normally, you cannot use numbers as part of a control sequence (or, command) name. This makes perfect sense and is good as it is. However, just to raise awareness to this, we provide a command here that changes the category codes of numbers 0–9 to 11, i. e. normal character. So they *can* be used in command names. However, this will break many packages, so do *not* expect anything to work! At least use it *after* all packages are loaded.

`\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` `\countwords` Counts every printed character (or word, respectively) that appears in anything that is a paragraph. Which is quite everything, in fact, *except* math mode! The total number

³en.wikipedia.org/wiki/Boustrophedon

⁴en.wikipedia.org/wiki/Rongorongo

of glyphs/words will be printed at the end of the log file/console output. For glyphs, also the number of use for every letter is printed separately.

\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

\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 WOBBBWOB BWOB 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.

\medievalumlaut Changes every lowercase umlaut into the corresponding vocale glyph with a small “e” glyph above it to show the origins of the german umlauts coming from ae, oe, ue. Text-variant may follow.

\nyanize A synonym for rainbowcolor.

\randomerror Just throws a random T_EX or L^AT_EX 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) T_EX user’s group meeting.

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

- \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 stream 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 ...
- \suppressonecharbreak** T_EX normally does not suppress a linebreak after words with only one character ("I", "a" etc.) This command suppresses line breaks. It is very similar to the code provided by the `imnpattypo` package and based on the same ideas. However, the code in `chickenize` has been written before the author knew `imnpattypo`, and the code differs a bit, might even be a bit faster. Well, test it!
- \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 it's functionality. So once you used `\chickenize`, it will chickenize the whole document up to `\unchickenize`. However, the paragraph in which `\unchickenize` appears, will *not* be chickenized. The same is true for all other manipulations. Take care that you don't `\un`-anything before activating it, as this will result in an error.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

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 L^AT_EX 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`!

Please don't fool around by mixing a `\text`-version with the non-`\text`-version. If you feel like it and are not pleased with the result, it is up to *you* to provide a stable and working solution.

2.4 Lua functions

As all features are implemented on the Lua side, you can use these functions independently. If you do so, please consult the corresponding subsections in the [implementation](#) part, because there are some variables that can be adapted to your need.

You can use the following code inside a `\directlua` statement or in a `luacode` environment (or the corresponding thing in your format):

```
luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize")
```

Replace `pre` by `post` to register into the post linebreak filter. The second argument (here: `chickenize`) specifies the function name; the available functions are listed below. You can supply a label as you like in the third argument. The fourth and last argument, which is omitted in the example, determines the order in which the functions in the callback are used. If you have no fancy stuff going on, you can safely use 1.

3 Options – How to Adjust It

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

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

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

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

`chickenstring = <table>` The string that is printed when using `\chickenize`. In fact, `chickenstring` is a table which allows for some more random action. To specify the default string, 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).

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

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

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` 27 (up to LuaTeX 0.80., it was 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(GLYPH,head)`, with the first argument giving the respective id of the nodes.⁹

⁹GLYPH here stands for the id that the glyph node type has. This number can be achieved by calling `GLYPH = nodeid("glyph")` which will result in the correct number independent of the LuaTeX version. We will use this substitute throughout this document.

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(GLYPH,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 T_EX file

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

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

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\ALT{%
5   \bgroup%
6   \fontspec{Latin Modern Sans}%
7   A%
8   \kern-.37em \raisebox{.7ex}{\scalebox{0.25}{L}}%
9   \kern-.0em \raisebox{-0.98ex}{T}%
10  \egroup%
11 }
12
13 \def\allownumberincommands{
14   \catcode`\0=11
15   \catcode`\1=11
16   \catcode`\2=11
17   \catcode`\3=11
18   \catcode`\4=11
19   \catcode`\5=11
20   \catcode`\6=11
21   \catcode`\7=11
22   \catcode`\8=11
23   \catcode`\9=11
24 }
25
26 \def\BEclerize{
27   \chickenize
28   \directlua{
29     chickenstring[1] = "noise noise"
30     chickenstring[2] = "atom noise"
31     chickenstring[3] = "shot noise"
```

```

32   chickenstring[4]   = "photon noise"
33   chickenstring[5]   = "camera noise"
34   chickenstring[6]   = "noising noise"
35   chickenstring[7]   = "thermal noise"
36   chickenstring[8]   = "electronic noise"
37   chickenstring[9]   = "spin noise"
38   chickenstring[10]  = "electron noise"
39   chickenstring[11]  = "Bogoliubov noise"
40   chickenstring[12]  = "white noise"
41   chickenstring[13]  = "brown noise"
42   chickenstring[14]  = "pink noise"
43   chickenstring[15]  = "bloch sphere"
44   chickenstring[16]  = "atom shot noise"
45   chickenstring[17]  = "nature physics"
46 }
47 }
48
49 \def\boustrophedon{
50   \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon,"boustrophedon")}}
51 \def\unboustrophedon{
52   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon")}}
53
54 \def\boustrophedonglyphs{
55   \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon_glyphs,"boustrophedonglyphs")}}
56 \def\unboustrophedonglyphs{
57   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon_glyphs")}}
58
59 \def\boustrophedoninverse{
60   \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon_inverse,"boustrophedoninverse")}}
61 \def\unboustrophedoninverse{
62   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon_inverse")}}
63
64 \def\bubblesort{
65   \directlua{luatexbase.add_to_callback("post_linebreak_filter",bubblesort,"bubblesort")}}
66 \def\unbubblesort{
67   \directlua{luatexbase.remove_from_callback("bubblesort","bubblesort")}}
68
69 \def\chickenize{
70   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize")
71     luatexbase.add_to_callback("start_page_number",
72       function() texio.write("[\"..status.total_pages) end ,\"cstartpage")
73     luatexbase.add_to_callback("stop_page_number",
74       function() texio.write(" chickens]") end,"cstoppage")
75     luatexbase.add_to_callback("stop_run",nicetext,"a nice text")
76   }
77 }

```

```

78 \def\unchickenize{
79   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","chickenize")}
80   luatexbase.remove_from_callback("start_page_number","cstartpage")
81   luatexbase.remove_from_callback("stop_page_number","cstoppage")}}
82
83 \def\coffeestainize{ %% to be implemented.
84   \directlua{}}
85 \def\uncoffeestainize{
86   \directlua{}}
87
88 \def\colorstretch{
89   \directlua{luatexbase.add_to_callback("post_linebreak_filter",colorstretch,"stretch_expansion")}
90 \def\uncolorstretch{
91   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","stretch_expansion")}}
92
93 \def\countglyphs{
94   \directlua{
95       counted_glyphs_by_code = {}
96       for i = 1,10000 do
97         counted_glyphs_by_code[i] = 0
98       end
99       glyphnumber = 0 spacenumber = 0
100      luatexbase.add_to_callback("post_linebreak_filter",countglyphs,"countglyphs")
101      luatexbase.add_to_callback("stop_run",printglyphnumber,"printglyphnumber")
102   }
103 }
104
105 \def\countwords{
106   \directlua{wordnumber = 0
107       luatexbase.add_to_callback("pre_linebreak_filter",countwords,"countwords")
108       luatexbase.add_to_callback("stop_run",printwordnumber,"printwordnumber")
109   }
110 }
111
112 \def\detectdoublewords{
113   \directlua{
114       luatexbase.add_to_callback("post_linebreak_filter",detectdoublewords,"detectdoublewords")
115       luatexbase.add_to_callback("stop_run",prindoublewords,"prindoublewords")
116   }
117 }
118
119 \def\dosomethingfunny{
120   %% should execute one of the "funny" commands, but randomly. So every compilation is complete.
121 }
122
123 \def\dubstepenize{

```

```

124 \chickenize
125 \directlua{
126     chickenstring[1] = "WOB"
127     chickenstring[2] = "WOB"
128     chickenstring[3] = "WOB"
129     chickenstring[4] = "BROOOAR"
130     chickenstring[5] = "WHEE"
131     chickenstring[6] = "WOB WOB WOB"
132     chickenstring[7] = "WAAAAAAAAAH"
133     chickenstring[8] = "duhduh duhduh duh"
134     chickenstring[9] = "BEEEEEEEEEW"
135     chickenstring[10] = "DEEEEEEEEEW"
136     chickenstring[11] = "EEEEEW"
137     chickenstring[12] = "boop"
138     chickenstring[13] = "buhdee"
139     chickenstring[14] = "bee bee"
140     chickenstring[15] = "BZZZRRRRRRRROOOOOOAAAAA"
141
142     chickenizefraction = 1
143 }
144 }
145 \let\dubstepize\dubstepenize
146
147 \def\gutenbergenize{ %% makes only sense when using LaTeX
148 \AtBeginDocument{
149     \let\grqq\relax\let\glqq\relax
150     \let\frqq\relax\let\flqq\relax
151     \let\grq\relax\let\glq\relax
152     \let\frq\relax\let\flq\relax
153 %
154     \gdef\footnote##1{}
155     \gdef\cite##1{}\gdef\parencite##1{}
156     \gdef\Cite##1{}\gdef\Parencite##1{}
157     \gdef\cites##1{}\gdef\parencites##1{}
158     \gdef\Cites##1{}\gdef\Parencites##1{}
159     \gdef\footcite##1{}\gdef\footcitetext##1{}
160     \gdef\footcites##1{}\gdef\footcitetexts##1{}
161     \gdef\textcite##1{}\gdef\Textcite##1{}
162     \gdef\textcites##1{}\gdef\Textcites##1{}
163     \gdef\smartcites##1{}\gdef\Smartcites##1{}
164     \gdef\supercite##1{}\gdef\supercites##1{}
165     \gdef\autocite##1{}\gdef\Autocite##1{}
166     \gdef\autocites##1{}\gdef\Autocites##1{}
167     %% many, many missing ... maybe we need to tackle the underlying mechanism?
168 }
169 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",gutenbergenize_rq,"gutenbergenize")

```



```

170 }
171
172 \def\hammertime{
173   \global\let\n\relax
174   \directlua{hammerfirst = true
175             luatexbase.add_to_callback("pre_linebreak_filter",hammertime,"hammertime")}}
176 \def\unhammertime{
177   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","hammertime")}}
178
179 % \def\itsame{
180 %   \directlua{drawmario}} %%% does not exist
181
182 \def\kernmanipulate{
183   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",kernmanipulate,"kernmanipulate")}}
184 \def\unkernmanipulate{
185   \directlua{lutaexbase.remove_from_callback("pre_linebreak_filter",kernmanipulate)}}
186
187 \def\leetspeak{
188   \directlua{luatexbase.add_to_callback("post_linebreak_filter",leet,"1337")}}
189 \def\unleetspeak{
190   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","1337")}}
191
192 \def\leftsideright#1{
193   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",leftsideright,"leftsideright")}}
194 \directlua{
195   leftsiderightindex = {#1}
196   leftsiderightarray = {}
197   for _,i in pairs(leftsiderightindex) do
198     leftsiderightarray[i] = true
199   end
200 }
201 }
202 \def\unleftsideright{
203   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","leftsideright")}}
204
205 \def\letterspaceadjust{
206   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",letterspaceadjust,"letterspaceadjust")}}
207 \def\unletterspaceadjust{
208   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","letterspaceadjust")}}
209
210 \def\listallcommands{
211   \directlua{
212     for name in pairs(tex.hashtokens()) do
213       print(name)
214     end}
215 }

```

```

216
217 \let\stealsheep\letterspaceadjust      %% synonym in honor of Paul
218 \let\unstealsheep\unletterspaceadjust
219 \let\returnsheep\unletterspaceadjust
220
221 \def\matrixize{
222   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",matrixize,"matrixize")}}
223 \def\unmatrixize{
224   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","matrixize")}}
225
226 \def\milkcow{      %% FIXME %% to be implemented
227   \directlua{}}
228 \def\unmilkcow{
229   \directlua{}}
230
231 \def\medievalumlaut{
232   \directlua{luatexbase.add_to_callback("post_linebreak_filter",medievalumlaut,"medievalumlaut")}}
233 \def\unmedievalumlaut{
234   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","medievalumlaut")}}
235
236 \def\pancakenize{
237   \directlua{luatexbase.add_to_callback("stop_run",pancaketext,"pancaketext")}}
238
239 \def\rainbowcolor{
240   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"rainbowcolor")
241     rainbowcolor = true}}
242 \def\unrainbowcolor{
243   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","rainbowcolor")
244     rainbowcolor = false}}
245 \let\nyanize\rainbowcolor
246 \let\unnyanize\unrainbowcolor
247
248 \def\randomcolor{
249   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"randomcolor")}}
250 \def\unrandomcolor{
251   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomcolor")}}
252
253 \def\randomerror{ %% FIXME
254   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomerror,"randomerror")}}
255 \def\unrandomerror{ %% FIXME
256   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomerror")}}
257
258 \def\randomfonts{
259   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomfonts,"randomfonts")}}
260 \def\unrandomfonts{
261   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomfonts")}}

```

```

262
263 \def\randomuclc{
264   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",randomuclc,"randomuclc")}}
265 \def\unrandomuclc{
266   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","randomuclc")}}
267
268 \let\rongorongonize\boustrophedoninverse
269 \let\unrongorongonize\unboustrophedoninverse
270
271 \def\scorpionize{
272   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",scorpionize_color,"scorpionize_color")}}
273 \def\unscorpionize{
274   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","scorpionize_color")}}
275
276 \def\spankmonkey{ %% to be implemented
277   \directlua{}}
278 \def\unspankmonkey{
279   \directlua{}}
280
281 \def\substitutewords{
282   \directlua{luatexbase.add_to_callback("process_input_buffer",substitutewords,"substitutewords")}}
283 \def\unsubstitutewords{
284   \directlua{luatexbase.remove_from_callback("process_input_buffer","substitutewords")}}
285
286 \def\addtosubstitutions#1#2{
287   \directlua{addtosubstitutions("#1","#2")}}
288 }
289
290 \def\suppressonecharbreak{
291   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",suppressonecharbreak,"suppressonecharbreak")}}
292 \def\unsuppressonecharbreak{
293   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","suppressonecharbreak")}}
294
295 \def\tabularasa{
296   \directlua{luatexbase.add_to_callback("post_linebreak_filter",tabularasa,"tabularasa")}}
297 \def\untabularasa{
298   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","tabularasa")}}
299
300 \def\tanjanize{
301   \directlua{luatexbase.add_to_callback("post_linebreak_filter",tanjanize,"tanjanize")}}
302 \def\untanjanize{
303   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","tanjanize")}}
304
305 \def\uppercasecolor{
306   \directlua{luatexbase.add_to_callback("post_linebreak_filter",uppercasecolor,"uppercasecolor")}}
307 \def\unuppercasecolor{

```

```

308 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","uppercasecolor")}}
309
310 \def\upsideown#1{
311 \directlua{luatexbase.add_to_callback("post_linebreak_filter",upsideown,"upsideown")}}
312 \directlua{
313     upsideownindex = {#1}
314     upsideownarray = {}
315     for _,i in pairs(upsideownindex) do
316         upsideownarray[i] = true
317     end
318 }
319 }
320 \def\unupsideown{
321 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","upsideown")}}
322
323 \def\unuppercasecolor{
324 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","upsideow")}}
325
326 \def\variantjustification{
327 \directlua{luatexbase.add_to_callback("post_linebreak_filter",variantjustification,"variantjust.
328 \def\unvariantjustification{
329 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","variantjustification")}}
330
331 \def\zebranize{
332 \directlua{luatexbase.add_to_callback("post_linebreak_filter",zebranize,"zebranize")}}
333 \def\unzebranize{
334 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","zebranize")}}

```

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

```

335 \newattribute\leetattr
336 \newattribute\letterspaceadjustattr
337 \newattribute\randcolorattr
338 \newattribute\randfontsattrib
339 \newattribute\randuclcattrib
340 \newattribute\tabularasattrib
341 \newattribute\uppercasecolorattr
342
343 \long\def\textleetspeak#1%
344   {\setluatexattribute\leetattr{42}#1\unsetluatexattribute\leetattr}
345
346 \long\def\textletterspaceadjust#1{
347   \setluatexattribute\letterspaceadjustattr{42}#1\unsetluatexattribute\letterspaceadjustattr
348   \directlua{
349     if (textletterspaceadjustactive) then else % -- if already active, do nothing
350       luatexbase.add_to_callback("pre_linebreak_filter",textletterspaceadjust,"textletterspaceadj
351     end

```

```

352     textletterspaceadjustactive = true           % -- set to active
353   }
354 }
355 \let\textlsa\textletterspaceadjust
356
357 \long\def\textrandomcolor#1%
358   {\setluatexattribute\randcolorattr{42}#1\unsetluatexattribute\randcolorattr}
359 \long\def\textrandomfonts#1%
360   {\setluatexattribute\randfontsaattr{42}#1\unsetluatexattribute\randfontsaattr}
361 \long\def\textrandomfontsa#1%
362   {\setluatexattribute\randfontsaattr{42}#1\unsetluatexattribute\randfontsaattr}
363 \long\def\textrandomuclc#1%
364   {\setluatexattribute\randuclcatr{42}#1\unsetluatexattribute\randuclcatr}
365 \long\def\texttabularasa#1%
366   {\setluatexattribute\tabularasaattr{42}#1\unsetluatexattribute\tabularasaattr}
367 \long\def\textuppercasecolor#1%
368   {\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.

```

369 \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.

```

370 \long\def\luadraw#1#2{%
371   \vbox to #1bp{%
372     \vfil
373     \luatexlatelua{pdf_print("q") #2 pdf_print("Q")}%
374   }%
375 }
376 \long\def\drawchicken{
377   \luadraw{90}{
378     kopf = {200,50} % Kopfmitte
379     kopf_rad = 20
380
381     d = {215,35} % Halsansatz
382     e = {230,10} %
383
384     korper = {260,-10}
385     korper_rad = 40
386
387     bein11 = {260,-50}
388     bein12 = {250,-70}
389     bein13 = {235,-70}
390
391     bein21 = {270,-50}
392     bein22 = {260,-75}

```

```

393 bein23 = {245,-75}
394
395 schnabel_oben = {185,55}
396 schnabel_vorne = {165,45}
397 schnabel_unten = {185,35}
398
399 flugel_vorne = {260,-10}
400 flugel_unten = {280,-40}
401 flugel_hinten = {275,-15}
402
403 sloppycircle(kopf,kopf_rad)
404 sloppyline(d,e)
405 sloppycircle(korper,korper_rad)
406 sloppyline(bein11,bein12) sloppyline(bein12,bein13)
407 sloppyline(bein21,bein22) sloppyline(bein22,bein23)
408 sloppyline(schnabel_vorne,schnabel_oben) sloppyline(schnabel_vorne,schnabel_unten)
409 sloppyline(flugel_vorne,flugel_unten) sloppyline(flugel_hinten,flugel_unten)
410 }
411 }

```

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.

```

412 \ProvidesPackage{chickenize}%
413 [2013/08/22 v0.2.1a chickenize package]
414 \input{chickenize}

```

9.1 Free Compliments

415

9.2 Definition of User-Level Macros

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

```

416 \iffalse
417 \DeclareDocumentCommand\includegraphics{0}{m}{
418   \fbox{Chicken}   %% actually, I'd love to draw an MP graph showing a chicken ...
419 }
420 %%% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
421 %% So far, you have to load pgfplots yourself.

```

```

422 %% As it is a mighty package, I don't want the user to force loading it.
423 \NewDocumentCommand\balmerpeak{G{}O{-4cm}}{
424 %% to be done using Lua drawing.
425 }
426 \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.

```

427
428 local nodenew = node.new
429 local nodecopy = node.copy
430 local nodetail = node.tail
431 local nodeinsertbefore = node.insert_before
432 local nodeinsertafter = node.insert_after
433 local noderemove = node.remove
434 local nodeid = node.id
435 local nodetraverseid = node.traverse_id
436 local nodeslide = node.slide
437
438 Hhead = nodeid("hhead")
439 RULE = nodeid("rule")
440 GLUE = nodeid("glue")
441 WHAT = nodeid("whatsit")
442 COL = node.subtype("pdf_colorstack")
443 PDF_LITERAL = node.subtype("pdf_literal")
444 GLYPH = nodeid("glyph")

```

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

```

445 color_push = nodenew(WHAT,COL)
446 color_pop = nodenew(WHAT,COL)
447 color_push.stack = 0
448 color_pop.stack = 0
449 color_push.command = 1
450 color_pop.command = 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.

```

451 chicken_pagenumbers = true
452

```

```

453 chickenstring = {}
454 chickenstring[1] = "chicken" -- chickenstring is a table, please remeber this!
455
456 chickenizefraction = 0.5
457 -- set this to a small value to fool somebody, or to see if your text has been read carefully. Th
458 chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing your
459
460 local match = unicode.utf8.match
461 chickenize_ignore_word = false

```

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

```

462 chickenize_real_stuff = function(i,head)
463     while ((i.next.id == GLYPH) or (i.next.id == 11) or (i.next.id == 7) or (i.next.id == 0)) do
464         i.next = i.next.next
465     end
466
467     chicken = {} -- constructing the node list.
468
469 -- Should this be done only once? No, otherwise we lose the freedom to change the string in-docum
470 -- But it could be done only once each paragraph as in-paragraph changes are not possible!
471
472     chickenstring_tmp = chickenstring[math.random(1,#chickenstring)]
473     chicken[0] = nodenew(GLYPH,1) -- only a dummy for the loop
474     for i = 1,string.len(chickenstring_tmp) do
475         chicken[i] = nodenew(GLYPH,1)
476         chicken[i].font = font.current()
477         chicken[i-1].next = chicken[i]
478     end
479
480     j = 1
481     for s in string.utfvalues(chickenstring_tmp) do
482         local char = unicode.utf8.char(s)
483         chicken[j].char = s
484         if match(char,"%s") then
485             chicken[j] = nodenew(10)
486             chicken[j].spec = nodenew(47)
487             chicken[j].spec.width = space
488             chicken[j].spec.shrink = shrink
489             chicken[j].spec.stretch = stretch
490         end
491         j = j+1
492     end
493
494     nodeslide(chicken[1])
495     lang.hyphenate(chicken[1])
496     chicken[1] = node.kerning(chicken[1]) -- FIXME: does not work
497     chicken[1] = node.ligaturing(chicken[1]) -- dito

```



```

498
499     nodeinsertbefore(head,i,chicken[1])
500     chicken[1].next = chicken[2] -- seems to be necessary ... to be fixed
501     chicken[string.len(chickenstring_tmp)].next = i.next
502
503     -- shift lowercase latin letter to uppercase if the original input was an uppercase
504     if (chickenize_capital and (chicken[1].char > 96 and chicken[1].char < 123)) then
505         chicken[1].char = chicken[1].char - 32
506     end
507
508     return head
509 end
510
511 chickenize = function(head)
512     for i in nodetraverseid(GLYPH,head) do --find start of a word
513         -- Random determination of the chickenization of the next word:
514         if math.random() > chickenizefraction then
515             chickenize_ignore_word = true
516         elseif chickencount then
517             chicken_substitutions = chicken_substitutions + 1
518         end
519
520         if (chickenize_ignore_word == false) then -- normal case: at the beginning of a word, we jump
521             if (i.char > 64 and i.char < 91) then chickenize_capital = true else chickenize_capital = false
522             head = chickenize_real_stuff(i,head)
523         end
524
525         -- At the end of the word, the ignoring is reset. New chance for everyone.
526         if not((i.next.id == GLYPH) or (i.next.id == 7) or (i.next.id == 22) or (i.next.id == 11)) then
527             chickenize_ignore_word = false
528         end
529     end
530     return head
531 end
532

```

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)

```

533 local separator      = string.rep("=", 28)
534 local texiowrite_nl = texio.write_nl
535 nicetext = function()
536     texiowrite_nl("Output written on "..tex.jobname.."pdf ("..status.total_pages.." chicken,".." eg
537     texiowrite_nl(" ")
538     texiowrite_nl(separator)
539     texiowrite_nl("Hello my dear user,")
540     texiowrite_nl("good job, now go outside and enjoy the world!")
541     texiowrite_nl(" ")

```

```

542 texiowrite_nl("And don't forget to feed your chicken!")
543 texiowrite_nl(separator .. "\n")
544 if chickencount then
545     texiowrite_nl("There were "..chicken_substitutions.." substitutions made.")
546     texiowrite_nl(separator)
547 end
548 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:

```

549 boustrophedon = function(head)
550     rot = node.new(8,PDF_LITERAL)
551     rot2 = node.new(8,PDF_LITERAL)
552     odd = true
553     for line in node.traverse_id(0,head) do
554         if odd == false then
555             w = line.width/65536*0.99625 -- empirical correction factor (?)
556             rot.data = "-1 0 0 1 "..w.." 0 cm"
557             rot2.data = "-1 0 0 1"..-w.." 0 cm"
558             line.head = node.insert_before(line.head,line.head,nodecopy(rot))
559             nodeinsertafter(line.head,nodetail(line.head),nodecopy(rot2))
560             odd = true
561         else
562             odd = false
563         end
564     end
565     return head
566 end

```

Glyphwise rotation:

```

567 boustrophedon_glyphs = function(head)
568     odd = false
569     rot = nodenew(8,PDF_LITERAL)
570     rot2 = nodenew(8,PDF_LITERAL)
571     for line in nodetraverseid(0,head) do
572         if odd==true then
573             line.dir = "TRT"
574             for g in nodetraverseid(GLYPH,line.head) do
575                 w = -g.width/65536*0.99625
576                 rot.data = "-1 0 0 1 " .. w .." 0 cm"
577                 rot2.data = "-1 0 0 1 " .. -w .." 0 cm"
578                 line.head = node.insert_before(line.head,g,nodecopy(rot))
579                 nodeinsertafter(line.head,g,nodecopy(rot2))

```

```

580     end
581     odd = false
582     else
583         line.dir = "TLT"
584         odd = true
585     end
586 end
587 return head
588 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.

```

589 boustrophedon_inverse = function(head)
590   rot = node.new(8,PDF_LITERAL)
591   rot2 = node.new(8,PDF_LITERAL)
592   odd = true
593   for line in node.traverse_id(0,head) do
594     if odd == false then
595 texio.write_nl(line.height)
596       w = line.width/65536*0.99625 -- empirical correction factor (?)
597       h = line.height/65536*0.99625
598       rot.data = "-1 0 0 -1 "..w.." "..h.." cm"
599       rot2.data = "-1 0 0 -1"..-w.." "..0.5*h.." cm"
600       line.head = node.insert_before(line.head,line.head,node.copy(rot))
601       node.insert_after(line.head,node.tail(line.head),node.copy(rot2))
602       odd = true
603     else
604       odd = false
605     end
606   end
607   return head
608 end

```

10.3 bubblesort

```

609 function bubblesort(head)
610   for line in nodetraverseid(0,head) do
611     for glyph in nodetraverseid(GLYPH,line.head) do
612
613     end
614   end
615   return head
616 end

```

10.4 countglyphs

Counts the glyphs in your document. Where “glyph” means every printed character in everything that is a paragraph – formulas do *not* work! Captions of floats etc. also will *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 document, but only after the hyphenation it is possible to count the real number of printed characters – where the hyphen does count.

Not only the total number of glyphs is recorded, but also the number of glyphs by character code. By this, you know exactly how many “a” or “ß” you used. A feature of category “completely useless”.

Spaces are also counted, but only spaces between glyphs in the output (i. e. nothing at the end/beginning of the lines), excluding indentation.

This function will (maybe, upon request) be extended to allow counting of whatever you want.

Take care: This will slow down the compilation extremely, by about a factor of 2! Only use for playing around or counting a final version of your document!

```
617 countglyphs = function(head)
618   for line in nodetraverseid(0,head) do
619     for glyph in nodetraverseid(GLYPH,line.head) do
620       glyphnumber = glyphnumber + 1
621       if (glyph.next.next) then
622         if (glyph.next.id == 10) and (glyph.next.next.id == GLYPH) then
623           spacenumber = spacenumber + 1
624         end
625         counted_glyphs_by_code[glyph.char] = counted_glyphs_by_code[glyph.char] + 1
626       end
627     end
628   end
629   return head
630 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? ...

```
631 printglyphnumber = function()
632   texiowrite_nl("\nNumber of glyphs by character code (only up to 127):")
633   for i = 1,127 do --%% FIXME: should allow for more characters, but cannot be printed to console
634     texiowrite_nl(string.char(i)..": " ..counted_glyphs_by_code[i])
635   end
636
637   texiowrite_nl("\nTotal number of glyphs in this document: " ..glyphnumber)
638   texiowrite_nl("Number of spaces in this document: " ..spacenumber)
639   texiowrite_nl("Glyphs plus spaces: " ..glyphnumber+spacenumber.." \n")
640 end
```

10.5 countwords

Counts the number of words in the document. The function works directly before the line breaking, so all macros are expanded. A “word” then is everything that is between two spaces before paragraph formatting. The beginning of a paragraph is a word, and the last word of a paragraph is accounted for by explicit increasing the counter, as no space token follows.

```
641 countwords = function(head)
```

```

642 for glyph in nodetraverseid(GLYPH,head) do
643   if (glyph.next.id == 10) then
644     wordnumber = wordnumber + 1
645   end
646 end
647 wordnumber = wordnumber + 1 -- add 1 for the last word in a paragraph which is not found otherwise
648 return head
649 end

```

Printing is done at the end of the compilation in the `stop_run` callback:

```

650 printwordnumber = function()
651   texio.write_nl("\nNumber of words in this document: "..wordnumber)
652 end

```

10.6 detectdoublewords

```

653 %% FIXME: Does this work? ...
654 function detectdoublewords(head)
655   prevlastword = {} -- array of numbers representing the glyphs
656   prevfirstword = {}
657   newlastword = {}
658   newfirstword = {}
659   for line in nodetraverseid(0,head) do
660     for g in nodetraverseid(GLYPH,line.head) do
661       texio.write_nl("next glyph",#newfirstword+1)
662       newfirstword[#newfirstword+1] = g.char
663       if (g.next.id == 10) then break end
664     end
665     texio.write_nl("nfw:"..#newfirstword)
666   end
667 end
668
669 function printdoublewords()
670   texio.write_nl("finished")
671 end

```

10.7 guttenbergenize

A function in honor of the German politician Guttenberg.¹⁰ 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.

¹⁰Thanks to Jasper for bringing me to this idea!

10.7.1 guttenbergenize – preliminaries

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

```
672 local quotestrings = {
673   [171] = true, [172] = true,
674   [8216] = true, [8217] = true, [8218] = true,
675   [8219] = true, [8220] = true, [8221] = true,
676   [8222] = true, [8223] = true,
677   [8248] = true, [8249] = true, [8250] = true,
678 }
```

10.7.2 guttenbergenize – the function

```
679 guttenbergenize_rq = function(head)
680   for n in nodetraverseid(nodeid"glyph",head) do
681     local i = n.char
682     if quotestrings[i] then
683       noderemove(head,n)
684     end
685   end
686   return head
687 end
```

10.8 hammertime

This is a completely useless function. It just prints STOP! – HAMMERTIME at the beginnig of the first paragraph after \hammertime, and “U can’t touch this” for every following one. As the function writes to the terminal, you have to be sure that your terminal is line-buffered and not block-buffered. Compare the explanation by Taco on the LuaTeX mailing list.¹¹

```
688 hammertimedelay = 1.2
689 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
690 hammertime = function(head)
691   if hammerfirst then
692     texiowrite_nl(htime_separator)
693     texiowrite_nl("=====STOP!=====\\n")
694     texiowrite_nl(htime_separator .. "\\n\\n\\n")
695     os.sleep (hammertimedelay*1.5)
696     texiowrite_nl(htime_separator .. "\\n")
697     texiowrite_nl("=====HAMMERTIME=====\\n")
698     texiowrite_nl(htime_separator .. "\\n\\n")
699     os.sleep (hammertimedelay)
700     hammerfirst = false
701   else
702     os.sleep (hammertimedelay)
703     texiowrite_nl(htime_separator)
```

¹¹<http://tug.org/pipermail/luatex/2011-November/003355.html>

```

704     texiowrite_nl("====U can't touch this!====\n")
705     texiowrite_nl(htime_separator .. "\n\n")
706     os.sleep (hammertimedelay*0.5)
707 end
708 return head
709 end

```

10.9 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.

```

710 itsame = function()
711 local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
712 color = "1 .6 0"
713 for i = 6,9 do mr(i,3) end
714 for i = 3,11 do mr(i,4) end
715 for i = 3,12 do mr(i,5) end
716 for i = 4,8 do mr(i,6) end
717 for i = 4,10 do mr(i,7) end
718 for i = 1,12 do mr(i,11) end
719 for i = 1,12 do mr(i,12) end
720 for i = 1,12 do mr(i,13) end
721
722 color = ".3 .5 .2"
723 for i = 3,5 do mr(i,3) end mr(8,3)
724 mr(2,4) mr(4,4) mr(8,4)
725 mr(2,5) mr(4,5) mr(5,5) mr(9,5)
726 mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
727 for i = 3,8 do mr(i,8) end
728 for i = 2,11 do mr(i,9) end
729 for i = 1,12 do mr(i,10) end
730 mr(3,11) mr(10,11)
731 for i = 2,4 do mr(i,15) end for i = 9,11 do mr(i,15) end
732 for i = 1,4 do mr(i,16) end for i = 9,12 do mr(i,16) end
733
734 color = "1 0 0"
735 for i = 4,9 do mr(i,1) end
736 for i = 3,12 do mr(i,2) end
737 for i = 8,10 do mr(5,i) end
738 for i = 5,8 do mr(i,10) end
739 mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
740 for i = 4,9 do mr(i,12) end
741 for i = 3,10 do mr(i,13) end
742 for i = 3,5 do mr(i,14) end
743 for i = 7,10 do mr(i,14) end
744 end

```

10.10 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.

```
745 chickenkernamount = 0
746 chickeninvertkerning = false
747
748 function kernmanipulate (head)
749   if chickeninvertkerning then -- invert the kerning
750     for n in nodetraverseid(11,head) do
751       n.kern = -n.kern
752     end
753   else -- if not, set it to the given value
754     for n in nodetraverseid(11,head) do
755       n.kern = chickenkernamount
756     end
757   end
758   return head
759 end
```

10.11 leetspeak

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

```
760 leetspeak_onlytext = false
761 leetteble = {
762   [101] = 51, -- E
763   [105] = 49, -- I
764   [108] = 49, -- L
765   [111] = 48, -- O
766   [115] = 53, -- S
767   [116] = 55, -- T
768
769   [101-32] = 51, -- e
770   [105-32] = 49, -- i
771   [108-32] = 49, -- l
772   [111-32] = 48, -- o
773   [115-32] = 53, -- s
774   [116-32] = 55, -- t
775 }
```

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

```
776 leet = function(head)
777   for line in nodetraverseid(Hhead,head) do
```



```

778   for i in nodetraverseid(GLYPH,line.head) do
779     if not leetspeak_onlytext or
780       node.has_attribute(i,luatexbase.attributes.leetattr)
781     then
782       if leettable[i.char] then
783         i.char = leettable[i.char]
784       end
785     end
786   end
787 end
788 return head
789 end

```

10.12 leftsideright

This function mirrors each glyph given in the array of leftsiderightarray horizontally.

```

790 leftsideright = function(head)
791   local factor = 65536/0.99626
792   for n in nodetraverseid(GLYPH,head) do
793     if (leftsiderightarray[n.char]) then
794       shift = nodenew(8,PDF_LITERAL)
795       shift2 = nodenew(8,PDF_LITERAL)
796       shift.data = "q -1 0 0 1 " .. n.width/factor .. " 0 cm"
797       shift2.data = "Q 1 0 0 1 " .. n.width/factor .. " 0 cm"
798       nodeinsertbefore(head,n,shift)
799       nodeinsertafter(head,n,shift2)
800     end
801   end
802   return head
803 end

```

10.13 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>

10.13.1 setup of variables

```

804 local letterspace_glue = nodenew(nodeid"glue")
805 local letterspace_spec = nodenew(nodeid"glue_spec")
806 local letterspace_pen = nodenew(nodeid"penalty")
807

```

```

808 letterspace_spec.width    = tex.sp"0pt"
809 letterspace_spec.stretch = tex.sp"0.05pt"
810 letterspace_glue.spec     = letterspace_spec
811 letterspace_pen.penalty   = 10000

```

10.13.2 function implementation

```

812 letterspaceadjust = function(head)
813   for glyph in nodetraverseid(nodeid"glyph", head) do
814     if glyph.prev and (glyph.prev.id == nodeid"glyph" or glyph.prev.id == nodeid"disc" or glyph.p
815       local g = nodecopy(letterspace_glue)
816       nodeinsertbefore(head, glyph, g)
817       nodeinsertbefore(head, g, nodecopy(letterspace_pen))
818     end
819   end
820   return head
821 end

```

10.13.3 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!

```

822 textletterspaceadjust = function(head)
823   for glyph in nodetraverseid(nodeid"glyph", head) do
824     if node.has_attribute(glyph, luatexbase.attributes.letterspaceadjustattr) then
825       if glyph.prev and (glyph.prev.id == node.id"glyph" or glyph.prev.id == node.id"disc" or gly
826         local g = node.copy(letterspace_glue)
827         nodeinsertbefore(head, glyph, g)
828         nodeinsertbefore(head, g, nodecopy(letterspace_pen))
829       end
830     end
831   end
832   luatexbase.remove_from_callback("pre_linebreak_filter", "textletterspaceadjust")
833   return head
834 end

```

10.14 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.

```

835 matrixize = function(head)
836   x = {}
837   s = nodenew(nodeid"disc")
838   for n in nodetraverseid(nodeid"glyph", head) do
839     j = n.char
840     for m = 0,7 do -- stay ASCII for now

```

```

841     x[7-m] = nodecopy(n) -- to get the same font etc.
842
843     if (j / (2^(7-m)) < 1) then
844         x[7-m].char = 48
845     else
846         x[7-m].char = 49
847         j = j-(2^(7-m))
848     end
849     nodeinsertbefore(head,n,x[7-m])
850     nodeinsertafter(head,x[7-m],nodecopy(s))
851 end
852 noderemove(head,n)
853 end
854 return head
855 end

```

10.15 medievalumlaut

Changes the umlauts ä, ö, ü into a, o, u with an e as an accent. The exact position of the e is adapted for each glyph, but that is only tested with one font. Other fonts might f*ck up everything.

For this, we define node representing the e (which then is copied every time) and two nodes that shift the e to where it belongs by using pdf matrix-nodes. An additional kern node shifts the space that the e took back so that everything ends up in the right place. All this happens in the `post_linebreak_filter` to enable normal hyphenation and line breaking. Well, `pre_linebreak_filter` would also have done ...

```

856 medievalumlaut = function(head)
857   local factor = 65536/0.99626
858   local org_e_node = nodenew(GLYPH)
859   org_e_node.char = 101
860   for line in nodetraverseid(0,head) do
861     for n in nodetraverseid(GLYPH,line.head) do
862       if (n.char == 228 or n.char == 246 or n.char == 252) then
863         e_node = nodecopy(org_e_node)
864         e_node.font = n.font
865         shift = nodenew(8,PDF_LITERAL)
866         shift2 = nodenew(8,PDF_LITERAL)
867         shift2.data = "Q 1 0 0 1 " .. e_node.width/factor .. " 0 cm"
868         nodeinsertafter(head,n,e_node)
869
870         nodeinsertbefore(head,e_node,shift)
871         nodeinsertafter(head,e_node,shift2)
872
873         x_node = nodenew(11)
874         x_node.kern = -e_node.width
875         nodeinsertafter(head,shift2,x_node)
876       end
877     end

```

```

878     if (n.char == 228) then -- ä
879         shift.data = "q 0.5 0 0 0.5 " ..
880             -n.width/factor*0.85 .. " " .. n.height/factor*0.75 .. " cm"
881         n.char = 97
882     end
883     if (n.char == 246) then -- ö
884         shift.data = "q 0.5 0 0 0.5 " ..
885             -n.width/factor*0.75 .. " " .. n.height/factor*0.75 .. " cm"
886         n.char = 111
887     end
888     if (n.char == 252) then -- ü
889         shift.data = "q 0.5 0 0 0.5 " ..
890             -n.width/factor*0.75 .. " " .. n.height/factor*0.75 .. " cm"
891         n.char = 117
892     end
893 end
894 end
895 return head
896 end

```

10.16 pancakenize

```

897 local separator      = string.rep("=", 28)
898 local texiowrite_nl = texio.write_nl
899 pancaketext = function()
900     texiowrite_nl("Output written on "..tex.jobname.." pdf ("..status.total_pages.." chicken,".." eg
901     texiowrite_nl(" ")
902     texiowrite_nl(separator)
903     texiowrite_nl("Soo ... you decided to use \\pancakenize.")
904     texiowrite_nl("That means you owe me a pancake!")
905     texiowrite_nl(" ")
906     texiowrite_nl("(This goes by document, not compilation.)")
907     texiowrite_nl(separator.."\\n\\n")
908     texiowrite_nl("Looking forward for my pancake! :)")
909     texiowrite_nl("\\n\\n")
910 end

```

10.17 randomerror

10.18 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.

```

911 randomfontslower = 1
912 randomfontsupper = 0
913 %
914 randomfonts = function(head)

```

```

915 local rfub
916 if randomfontsupper > 0 then -- fixme: this should be done only once, no? Or at every paragraph
917     rfub = randomfontsupper -- user-specified value
918 else
919     rfub = font.max() -- or just take all fonts
920 end
921 for line in nodetraverseid(Hhead,head) do
922     for i in nodetraverseid(GLYPH,line.head) do
923         if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattrib) then
924             i.font = math.random(randomfontslower,rfub)
925         end
926     end
927 end
928 return head
929 end

```

10.19 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```

930 uclcratio = 0.5 -- ratio between uppercase and lower case
931 randomuclc = function(head)
932     for i in nodetraverseid(GLYPH,head) do
933         if not(randomuclc_onlytext) or node.has_attribute(i,luatexbase.attributes.randuclcattrib) then
934             if math.random() < uclcratio then
935                 i.char = tex.uccode[i.char]
936             else
937                 i.char = tex.lccode[i.char]
938             end
939         end
940     end
941     return head
942 end

```

10.20 randomchars

```

943 randomchars = function(head)
944     for line in nodetraverseid(Hhead,head) do
945         for i in nodetraverseid(GLYPH,line.head) do
946             i.char = math.floor(math.random()*512)
947         end
948     end
949     return head
950 end

```

10.21 randomcolor and rainbowcolor

10.21.1 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.

```
951 randomcolor_grey = false
952 randomcolor_onlytext = false --switch between local and global colorization
953 rainbowcolor = false
954
955 grey_lower = 0
956 grey_upper = 900
957
958 Rgb_lower = 1
959 rGb_lower = 1
960 rgB_lower = 1
961 Rgb_upper = 254
962 rGb_upper = 254
963 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.

```
964 rainbow_step = 0.005
965 rainbow_Rgb = 1-rainbow_step -- we start in the red phase
966 rainbow_rGb = rainbow_step -- values x must always be  $0 < x < 1$ 
967 rainbow_rgB = rainbow_step
968 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.

```
969 randomcolorstring = function()
970   if randomcolor_grey then
971     return (0.001*math.random(grey_lower, grey_upper)).. " g"
972   elseif rainbowcolor then
973     if rainind == 1 then -- red
974       rainbow_rGb = rainbow_rGb + rainbow_step
975       if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
976     elseif rainind == 2 then -- yellow
977       rainbow_Rgb = rainbow_Rgb - rainbow_step
978       if rainbow_Rgb <= rainbow_step then rainind = 3 end
979     elseif rainind == 3 then -- green
980       rainbow_rgB = rainbow_rgB + rainbow_step
981       rainbow_rGb = rainbow_rGb - rainbow_step
982       if rainbow_rGb <= rainbow_step then rainind = 4 end
983     elseif rainind == 4 then -- blue
984       rainbow_Rgb = rainbow_Rgb + rainbow_step
985       if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
986     else -- purple
987       rainbow_rgB = rainbow_rgB - rainbow_step
```

```

988     if rainbow_rgb <= rainbow_step then rainind = 1 end
989   end
990   return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rgb.." rg"
991 else
992   Rgb = math.random(Rgb_lower,Rgb_upper)/255
993   rGb = math.random(rGb_lower,rGb_upper)/255
994   rgb = math.random(rgb_lower,rgb_upper)/255
995   return Rgb.." "..rGb.." "..rgb.." .." rg"
996 end
997 end

```

10.21.2 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.

```

998 randomcolor = function(head)
999   for line in nodetraverseid(0,head) do
1000     for i in nodetraverseid(GLYPH,line.head) do
1001       if not(randomcolor_onlytext) or
1002         (node.has_attribute(i,luatexbase.attributes.randcolorattr))
1003       then
1004         color_push.data = randomcolorstring() -- color or grey string
1005         line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
1006         nodeinsertafter(line.head,i,nodecopy(color_pop))
1007       end
1008     end
1009   end
1010   return head
1011 end

```

10.22 randomerror

1012 %

10.23 rickroll

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

1013 %

10.24 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.

```

1014 substitutewords_strings = {}
1015
1016 addtosubstitutions = function(input,output)
1017   substitutewords_strings[#substitutewords_strings + 1] = {}
1018   substitutewords_strings[#substitutewords_strings][1] = input
1019   substitutewords_strings[#substitutewords_strings][2] = output
1020 end
1021
1022 substitutewords = function(head)
1023   for i = 1,#substitutewords_strings do
1024     head = string.gsub(head,substitutewords_strings[i][1],substitutewords_strings[i][2])
1025   end
1026   return head
1027 end

```

10.25 suppressonecharbreak

We rush through the node list before line breaking takes place and insert large penalties for breaks after single glyphs. To keep the code as small, simple and fast as possible, we `traverse_id` over spaces and see whether the next `.next` node is also a space. This might not be the best and most universal way of doing it, but the simplest. The penalty is not created newly each time, but copied – no significant speed gain, however.

```

1028 suppressonecharbreakpenaltynode = node.new(12)
1029 suppressonecharbreakpenaltynode.penalty = 10000
1030 function suppressonecharbreak(head)
1031   for i in node.traverse_id(10,head) do
1032     if ((i.next) and (i.next.next.id == 10)) then
1033       pen = node.copy(suppressonecharbreakpenaltynode)
1034       node.insert_after(head,i.next,pen)
1035     end
1036   end
1037
1038   return head
1039 end

```

10.26 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.

```

1040 tabularasa_onlytext = false
1041

```



```

1042 tabularasa = function(head)
1043   local s = nodenew(nodeid"kern")
1044   for line in nodetraverseid(nodeid"hlist",head) do
1045     for n in nodetraverseid(nodeid"glyph",line.head) do
1046       if not(tabularasa_onlytext) or node.has_attribute(n,luatexbase.attributes.tabularasaattr) then
1047         s.kern = n.width
1048         nodeinsertafter(line.list,n,nodecopy(s))
1049         line.head = noderemove(line.list,n)
1050       end
1051     end
1052   end
1053   return head
1054 end

```

10.27 tanjanize

```

1055 tanjanize = function(head)
1056   local s = nodenew(nodeid"kern")
1057   local m = nodenew(GLYPH,1)
1058   local use_letter_i = true
1059   scale = nodenew(8,PDF_LITERAL)
1060   scale2 = nodenew(8,PDF_LITERAL)
1061   scale.data = "0.5 0 0 0.5 0 0 cm"
1062   scale2.data = "2 0 0 2 0 0 cm"
1063
1064   for line in nodetraverseid(nodeid"hlist",head) do
1065     for n in nodetraverseid(nodeid"glyph",line.head) do
1066       mimicount = 0
1067       tmpwidth = 0
1068       while ((n.next.id == GLYPH) or (n.next.id == 11) or (n.next.id == 7) or (n.next.id == 0)) do
1069         n.next = n.next.next
1070         mimicount = mimicount + 1
1071         tmpwidth = tmpwidth + n.width
1072       end
1073
1074       mimi = {} -- constructing the node list.
1075       mimi[0] = nodenew(GLYPH,1) -- only a dummy for the loop
1076       for i = 1,string.len(mimicount) do
1077         mimi[i] = nodenew(GLYPH,1)
1078         mimi[i].font = font.current()
1079         if(use_letter_i) then mimi[i].char = 109 else mimi[i].char = 105 end
1080         use_letter_i = not(use_letter_i)
1081         mimi[i-1].next = mimi[i]
1082       end
1083     end
1084   end

```

```

1085 line.head = nodeinsertbefore(line.head,n,nodecopy(scale))
1086 nodeinsertafter(line.head,n,nodecopy(scale2))
1087     s.kern = (tmpwidth*2-n.width)
1088     nodeinsertafter(line.head,n,nodecopy(s))
1089 end
1090 end
1091 return head
1092 end

```

10.28 uppercasecolor

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

```

1093 uppercasecolor_onlytext = false
1094
1095 uppercasecolor = function (head)
1096   for line in nodetraverseid(Hhead,head) do
1097     for upper in nodetraverseid(GLYPH,line.head) do
1098       if not(uppercasecolor_onlytext) or node.has_attribute(upper,luatexbase.attributes.uppercasecolor) then
1099         if ((upper.char > 64) and (upper.char < 91)) or
1100            ((upper.char > 57424) and (upper.char < 57451))) then -- for small caps! nice
1101           color_push.data = randomcolorstring() -- color or grey string
1102           line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
1103           nodeinsertafter(line.head,upper,nodecopy(color_pop))
1104         end
1105       end
1106     end
1107   end
1108   return head
1109 end

```

10.29 upsidedown

This function mirrors all glyphs given in the array upsidedownarray vertically.

```

1110 upsidedown = function(head)
1111   local factor = 65536/0.99626
1112   for line in nodetraverseid(Hhead,head) do
1113     for n in nodetraverseid(GLYPH,line.head) do
1114       if (upsidedownarray[n.char]) then
1115         shift = nodenew(8,PDF_LITERAL)
1116         shift2 = nodenew(8,PDF_LITERAL)
1117         shift.data = "q 1 0 0 -1 0 " .. n.height/factor .. " cm"
1118         shift2.data = "Q 1 0 0 1 " .. n.width/factor .. " 0 cm"
1119         nodeinsertbefore(head,n,shift)
1120         nodeinsertafter(head,n,shift2)
1121       end
1122     end
1123   end

```

```

1124 return head
1125 end

```

10.30 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!

10.30.1 colorstretch – preliminaries

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

```

1126 keeptext = true
1127 colorexpansion = true
1128
1129 colorstretch_coloroffset = 0.5
1130 colorstretch_colorange = 0.5
1131 chickenize_rule_bad_height = 4/5 -- height and depth of the rules
1132 chickenize_rule_bad_depth = 1/5
1133
1134
1135 colorstretchnumbers = true
1136 drawstretchthreshold = 0.1
1137 drawexpansionthreshold = 0.9

```

After these constants have been set, the function starts. It receives the vertical list of the typeset paragraph as `head`, and loops through all horizontal lists.

If font expansion should be shown (`colorexpansion == true`), then the first glyph node is determined and its width compared with the width of the unexpanded glyph. This gives a measure for the expansion factor and is translated into a grey scale.

```

1138 colorstretch = function (head)
1139   local f = font.getfont(font.current()).characters
1140   for line in nodetraverseid(Hhead,head) do
1141     local rule_bad = nodenew(RULE)
1142
1143     if colorexpansion then -- if also the font expansion should be shown
1144       local g = line.head
1145       while not(g.id == GLYPH) and (g.next) do g = g.next end -- find first glyph on line. If line
1146       if (g.id == GLYPH) then -- read width only if g is a glyph!
1147         exp_factor = g.width / f[g.char].width

```

```

1148     exp_color = colorstretch_coloroffset + (1-exp_factor)*10 .. " g"
1149     rule_bad.width = 0.5*line.width  -- we need two rules on each line!
1150   end
1151 else
1152   rule_bad.width = line.width  -- only the space expansion should be shown, only one rule
1153 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.

```

1154 rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bet
1155 rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth
1156
1157 local glue_ratio = 0
1158 if line.glue_order == 0 then
1159   if line.glue_sign == 1 then
1160     glue_ratio = colorstretch_colorrage * math.min(line.glue_set,1)
1161   else
1162     glue_ratio = -colorstretch_colorrage * math.min(line.glue_set,1)
1163   end
1164 end
1165 color_push.data = colorstretch_coloroffset + glue_ratio .. " g"
1166

```

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

```

1167 -- set up output
1168   local p = line.head
1169
1170 -- a rule to immitate kerning all the way back
1171   local kern_back = nodenew(RULE)
1172   kern_back.width = -line.width
1173
1174 -- if the text should still be displayed, the color and box nodes are inserted additionally
1175 -- and the head is set to the color node
1176   if keptext then
1177     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1178   else
1179     node.flush_list(p)
1180     line.head = nodecopy(color_push)
1181   end
1182   nodeinsertafter(line.head,line.head,rule_bad)  -- then the rule
1183   nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
1184   tmpnode = nodeinsertafter(line.head,line.head.next.next,kern_back)
1185
1186 -- then a rule with the expansion color
1187 if colorexansion then  -- if also the stretch/shrink of letters should be shown
1188   color_push.data = exp_color
1189   nodeinsertafter(line.head,tmpnode,nodecopy(color_push))

```

```

1190     nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
1191     nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
1192 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.

```

1193 if colorstretchnumbers then
1194     j = 1
1195     glue_ratio_output = {}
1196     for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
1197         local char = unicode.utf8.char(s)
1198         glue_ratio_output[j] = nodenew(GLYPH,1)
1199         glue_ratio_output[j].font = font.current()
1200         glue_ratio_output[j].char = s
1201         j = j+1
1202     end
1203     if math.abs(glue_ratio) > drawstretchthreshold then
1204         if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
1205         else color_push.data = "0 0.99 0 rg" end
1206     else color_push.data = "0 0 0 rg"
1207     end
1208
1209     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
1210     for i = 1,math.min(j-1,7) do
1211         nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])
1212     end
1213     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
1214 end -- end of stretch number insertion
1215 end
1216 return head
1217 end

```

dubstepize

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

```

1218

```

scorpionize

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

```

1219 function scorpionize_color(head)
1220     color_push.data = ".35 .55 .75 rg"
1221     nodeinsertafter(head,head,nodecopy(color_push))

```

```

1222 nodeinsertafter(head,node.tail(head),nodecopy(color_pop))
1223 return head
1224 end

```

10.31 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 ... (?)

```

1225 substlist = {}
1226 substlist[1488] = 64289
1227 substlist[1491] = 64290
1228 substlist[1492] = 64291
1229 substlist[1499] = 64292
1230 substlist[1500] = 64293
1231 substlist[1501] = 64294
1232 substlist[1512] = 64295
1233 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”).

```

1234 function variantjustification(head)
1235   math.randomseed(1)
1236   for line in nodetraverseid(nodeid"hhead",head) do
1237     if (line.glue_sign == 1 and line.glue_order == 0) then -- exclude the last line!
1238       substitutions_wide = {} -- we store all "expandable" letters of each line
1239       for n in nodetraverseid(nodeid"glyph",line.head) do
1240         if (substlist[n.char]) then
1241           substitutions_wide[#substitutions_wide+1] = n
1242         end
1243       end
1244       line.glue_set = 0 -- deactivate normal glue expansion
1245       local width = node.dimensions(line.head) -- check the new width of the line
1246       local goal = line.width
1247       while (width < goal and #substitutions_wide > 0) do
1248         x = math.random(#substitutions_wide) -- choose randomly a glyph to be substituted
1249         oldchar = substitutions_wide[x].char
1250         substitutions_wide[x].char = substlist[substitutions_wide[x].char] -- substitute by wide
1251         width = node.dimensions(line.head) -- check if the line is too wide
1252         if width > goal then substitutions_wide[x].char = oldchar break end -- substitute back if
1253         table.remove(substitutions_wide,x) -- if further substitutions have to be done,
1254       end

```

```

1255     end
1256 end
1257 return head
1258 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.

10.32 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`.

10.32.1 zebranize – preliminaries

```

1259 zebracolorarray = {}
1260 zebracolorarray_bg = {}
1261 zebracolorarray[1] = "0.1 g"
1262 zebracolorarray[2] = "0.9 g"
1263 zebracolorarray_bg[1] = "0.9 g"
1264 zebracolorarray_bg[2] = "0.1 g"

```

10.32.2 zebranize – the function

This code has to be revisited, it is ugly.

```

1265 function zebranize(head)
1266   zebracolor = 1
1267   for line in nodetraverseid(nodeid"hhead",head) do
1268     if zebracolor == #zebracolorarray then zebracolor = 0 end
1269     zebracolor = zebracolor + 1
1270     color_push.data = zebracolorarray[zebracolor]
1271     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1272     for n in nodetraverseid(nodeid"glyph",line.head) do
1273       if n.next then else
1274         nodeinsertafter(line.head,n,nodecopy(color_pull))
1275       end
1276     end
1277
1278     local rule_zebra = nodenew(RULE)
1279     rule_zebra.width = line.width
1280     rule_zebra.height = tex.baselineskip.width*4/5
1281     rule_zebra.depth = tex.baselineskip.width*1/5
1282

```

```
1283     local kern_back = nodenew(RULE)
1284     kern_back.width = -line.width
1285
1286     color_push.data = zebracolorarray_bg[zebracolor]
1287     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
1288     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1289     nodeinsertafter(line.head,line.head,kern_back)
1290     nodeinsertafter(line.head,line.head,rule_zebra)
1291 end
1292 return (head)
1293 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!

```

1294 --
1295 function pdf_print (...)
1296   for _, str in ipairs({...}) do
1297     pdf.print(str .. " ")
1298   end
1299   pdf.print("\n")
1300 end
1301
1302 function move (p)
1303   pdf_print(p[1],p[2],"m")
1304 end
1305
1306 function line (p)
1307   pdf_print(p[1],p[2],"l")
1308 end
1309
1310 function curve(p1,p2,p3)
1311   pdf_print(p1[1], p1[2],
1312             p2[1], p2[2],
1313             p3[1], p3[2], "c")
1314 end
1315
1316 function close ()
1317   pdf_print("h")
1318 end
1319
1320 function linewidth (w)
1321   pdf_print(w,"w")
1322 end
1323
1324 function stroke ()
1325   pdf_print("S")
1326 end
1327 --
1328

```

```

1329 function strictcircle(center,radius)
1330   local left = {center[1] - radius, center[2]}
1331   local lefttop = {left[1], left[2] + 1.45*radius}
1332   local leftbot = {left[1], left[2] - 1.45*radius}
1333   local right = {center[1] + radius, center[2]}
1334   local righttop = {right[1], right[2] + 1.45*radius}
1335   local rightbot = {right[1], right[2] - 1.45*radius}
1336
1337   move (left)
1338   curve (lefttop, righttop, right)
1339   curve (rightbot, leftbot, left)
1340 stroke()
1341 end
1342
1343 function disturb_point(point)
1344   return {point[1] + math.random()*5 - 2.5,
1345           point[2] + math.random()*5 - 2.5}
1346 end
1347
1348 function sloppycircle(center,radius)
1349   local left = disturb_point({center[1] - radius, center[2]})
1350   local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
1351   local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}
1352   local right = disturb_point({center[1] + radius, center[2]})
1353   local righttop = disturb_point({right[1], right[2] + 1.45*radius})
1354   local rightbot = disturb_point({right[1], right[2] - 1.45*radius})
1355
1356   local right_end = disturb_point(right)
1357
1358   move (right)
1359   curve (rightbot, leftbot, left)
1360   curve (lefttop, righttop, right_end)
1361   linewidth(math.random()+0.5)
1362   stroke()
1363 end
1364
1365 function sloppyline(start,stop)
1366   local start_line = disturb_point(start)
1367   local stop_line = disturb_point(stop)
1368   start = disturb_point(start)
1369   stop = disturb_point(stop)
1370   move(start) curve(start_line,stop_line,stop)
1371   linewidth(math.random()+0.5)
1372   stroke()
1373 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:

traversing Every function that is based on node traversing fails when boxes are involved – so far I have not implemented recursive calling of the functions. I list it here, as it is not really a bug – this package is meant to be as simple as possible!

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 thank Philipp Gesang who gave me many advices on the Lua code – which I still didn't have time to correct ...