



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

Paul Isambert

# CHICKENIZE

v0.2.9

Arno L. Trautmann  $\text{\textA}$

[arno.trautmann@gmx.de](mailto:arno.trautmann@gmx.de)

## How to read this document.

This is the documentation of the package `chickenize`. It allows manipulations of any Lua $\text{\TeX}$  document<sup>1</sup> 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 production document.

The table on the next page shortly informs you about some of your possibilities and provides links to the (documented) Lua functions. The  $\text{\TeX}$  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 only be considered stable and long-term compatible should it reach version 1.0.

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 © 2020 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'.

---

<sup>1</sup>The code is based on pure Lua $\text{\TeX}$  features, so don't even try to use it with any other  $\text{\TeX}$  flavour. The package is (partially) tested under plain Lua $\text{\TeX}$  and (fully) under Lua $\text{\LaTeX}$ . If you tried using it with Con $\text{\TeX}$ 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.<sup>2</sup> Of course, the label “complete nonsense” depends on what you are doing ... The links will take you to the source code, while a more complete list with explanations is given [further below](#).

---

### maybe useful functions

<a href="#">colorstretch</a>	shows grey boxes that visualise the badness and font expansion line-wise
<a href="#">letterspaceadjust</a>	improves the greyness by using a small amount of letterspacing
<a href="#">substitutewords</a>	replaces words by other words (chosen by the user)
<a href="#">variantjustification</a>	Justification by using glyph variants
<a href="#">suppressonecharbreak</a>	suppresses linebreaks after single-letter words

---

### less useful functions

<a href="#">boustrophedon</a>	invert every second line in the style of archaic greek texts
<a href="#">countglyphs</a>	counts the number of glyphs in the whole document
<a href="#">countwords</a>	counts the number of words in the whole document
<a href="#">leetspeak</a>	translates the (latin-based) input into 1337 5p34k
<a href="#">medievalumlaut</a>	changes each umlaut to normal glyph plus “e” above it: âôû
<a href="#">randomucl</a>	alternates randomly between uppercase and lowercase
<a href="#">rainbowcolor</a>	changes the color of letters slowly according to a rainbow
<a href="#">randomcolor</a>	prints every letter in a random color
<a href="#">tabularasa</a>	removes every glyph from the output and leaves an empty document
<a href="#">uppercasecolor</a>	makes every uppercase letter colored

---

### complete nonsense

<a href="#">chickenize</a>	replaces every word with “chicken” (or user-adjustable words)
<a href="#">drawchicken</a>	draws a nice chicken with random, “hand-sketch”-type lines
<a href="#">gutenbergize</a>	deletes every quote and footnotes
<a href="#">hammertime</a>	U can’t touch this!
<a href="#">italianize</a>	Mamma mia!!
<a href="#">italianizerandword</a>	Will put the word order in a sentence at random. (tbi)
<a href="#">kernmanipulate</a>	manipulates the kerning (tbi)
<a href="#">matrixize</a>	replaces every glyph by its ASCII value in binary code
<a href="#">randomerror</a>	just throws random (La)TeX errors at random times (tbi)
<a href="#">randomfonts</a>	changes the font randomly between every letter
<a href="#">randomchars</a>	randomizes the (letters of the) whole input

---

---

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

# Contents

<b>I</b>	<b>User Documentation</b>	<b>6</b>
1	How It Works	6
2	Commands – How You Can Use It	6
2.1	TeX Commands – Document Wide	6
2.2	How to Deactivate It	9
2.3	\text-Versions	9
2.4	Lua functions	9
3	Options – How to Adjust It	9
3.1	Options for Game of Chicken	11
<b>II</b>	<b>Tutorial</b>	<b>13</b>
4	Lua code	13
5	callbacks	13
5.1	How to use a callback	14
6	Nodes	14
7	Other things	15
<b>III</b>	<b>Implementation</b>	<b>16</b>
8	TeX file	16
8.1	allownumberincommands	16
8.2	drawchicken	26
8.3	drawcov	26
9	ETEX package	27
9.1	Free Compliments	27
9.2	Definition of User-Level Macros	28
10	Lua Module	28
10.1	chickenize	29
10.2	boustrophedon	31
10.3	bubblesort	33
10.4	countglyphs	33
10.5	countwords	34
10.6	detectdoublewords	34
10.7	francize	35

10.8	gamofchicken	35
10.9	gutenbergize	37
10.9.1	gutenbergize – preliminaries	38
10.9.2	gutenbergize – the function	38
10.10	hammertime	38
10.11	italianize	39
10.12	hammertime	41
10.13	itsame	42
10.14	kernmanipulate	43
10.15	leetspeak	43
10.16	leftsideright	44
10.17	letterspaceadjust	45
10.17.1	setup of variables	45
10.17.2	function implementation	45
10.17.3	textletterspaceadjust	45
10.18	matrixize	46
10.19	medievalumlaut	46
10.20	pancakenize	47
10.21	randomerror	48
10.22	randomfonts	48
10.23	randomucl	48
10.24	randomchars	49
10.25	randomcolor and rainbowcolor	49
10.25.1	randomcolor – preliminaries	49
10.25.2	randomcolor – the function	50
10.26	randomerror	50
10.27	relationship	51
10.28	rickroll	52
10.29	substitutewords	52
10.30	suppressonecharbreak	52
10.31	tabularasa	53
10.32	tanjanize	53
10.33	uppercasecolor	54
10.34	upsideup	54
10.35	colorstretch	55
10.35.1	colorstretch – preliminaries	55
10.36	variantjustification	58
10.37	zebranize	59
10.37.1	zebranize – preliminaries	59
10.37.2	zebranize – the function	60
<b>11</b>	<b>Drawing</b>	<b>61</b>
<b>12</b>	<b>Known Bugs and Fun Facts</b>	<b>64</b>
<b>13</b>	<b>To Do's</b>	<b>64</b>

<b>14 Literature</b>	<b>64</b>
<b>15 Thanks</b>	<b>65</b>

## Part I

# User Documentation

## 1 How It Works

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

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

## 2 Commands – How You Can Use It

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

### 2.1 $\TeX$ Commands – Document Wide

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

The effect of the commands can be influenced, not with arguments, but only via the `\chickenizesetup` described [below](#). The links provide here will bring you to the more relevant part of the implementation, i. e. either the  $\TeX$  code or the Lua code, depending on what is doing the main job. Mostly it's the Lua part.

**`\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.<sup>3</sup> 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 Rongorongo<sup>4</sup> similar style boustrophedon is available with `\boustrophedoninverse` or `\rongorongonize`, where subsequent lines are rotated by 180° instead of mirrored.

---

<sup>3</sup>[en.wikipedia.org/wiki/Boustrophedon](https://en.wikipedia.org/wiki/Boustrophedon)

<sup>4</sup>[en.wikipedia.org/wiki/Rongorongo](https://en.wikipedia.org/wiki/Rongorongo)

**\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 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 10<sup>th</sup> chicken is uppercase. However, the beginning of a sentence is not recognized automatically.<sup>5</sup>

**\drawchicken** Draws a chicken based on some low-level lua drawing code. Each stroke is parameterized with random numbers so the chicken will always look different.

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

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

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

**\explainbackslashes** A small list that gives hints on how many \ characters you actually need for a backslash. I’s supposed to be funny. At least my head thinks it’s funny. Inspired (and mostly copied from, actually) xkcd.

**\gameofchicken** This is a tentative implementation of Conway’s classic Game of Life. This is actually a rather powerful code with some choices for you. The game itself is played on a matrix in Lua and can be output either on the console (for quick checks) or in a pdf. The latter case needs a LaTeX document, and the packages geometry, placeat, and graphicx. You can choose which  $\LaTeX$  code represents the cells or you take the pre-defined – a ☹, of course! Additionally, there are anticells which is basically just a second set of cells. However, they can interact, and you have full control over the rules, i. e. how many neighbors a cell or anticell may need to be born, die, or stay alive, and what happens if cell and anticell collide. See below for parameters; all of them start with GOC for clarity.

**\gameoflife** Try it.

**\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.

---

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

- \nyanize** A synonym for `rainbowcolor`.
- \randomerror** Just throws a random  $\TeX$  or  $\LaTeX$  error at a random time during the compilation. I have quite no idea what this could be used for.
- \randomucl** Changes every character of the input into its uppercase or lowercase variant. Well, guess what the “random” means ...
- \randomfonts** Changes the font randomly for every character. If no parameters are given, all fonts that have been loaded are used, especially including math fonts.
- \randomcolor** Does what its name says.
- \rainbowcolor** Instead of random colors, this command causes the text color to change gradually according to the colors of a rainbow. Do not mix this with `randomcolor`, as that doesn’t make any sense.
- \relationship** Draws the relationship. A ship made of relations.
- \pancakenize** This is a dummy command that does nothing. However, every time you use it, you owe a pancake to the package author. You can either send it via mail or bring it to some (local)  $\TeX$  user’s group meeting.
- \substitutewords** You have to specify pairs of words by using `\addtosubstitutions{word1}{word2}`. Then call `\substitutewords` (or the other way round, doesn’t matter) and each occurrence of `word1` will be replaced by `word2`. You can add replacement pairs by repeated calls to `\addtosubstitutions`. Take care! This function works with the input 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**  $\TeX$  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 `impnatty` package and based on the same ideas. However, the code in `chickenize` has been written before the author knew `impnatty`, 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.<sup>6</sup>

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 have<sup>7</sup> 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.<sup>8</sup>

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?

---

<sup>6</sup>Which is so far not catchable due to missing functionality in `luatexbase`.

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

<sup>8</sup>On a 500 pages text-only  $\text{\LaTeX}$  document the dilation is on the order of 10% with `\textrandomcolor`, but other manipulations can take much more time. However, you are not supposed to make such long documents with `chickenize`!

However, `\chickenizesetup` is a macro on the  $\TeX$  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`).

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

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

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

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

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

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

`colorexpan = <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?)

### 3.1 Options for Game of Chicken

This deserves a separate section since there are some more options and they need some explanation. So here goes the parameters for the GOC:

`GOCrule_live = <{int,int,...}> {2,3}` This gives the number of neighbors for an existing cell to keep it alive. This is a list, so you can say `\chickenizesetup{GOCrule_live = {2,3,7}}` or similar.

`GOCrule_spawn = <{int,int,...}> {3}` The number of neighbors to spawn a new cell.

`GOCrule_antilive = <int> 2,3` The number of neighbors to keep an anticell alive.

`GOCrule_antispawn = <int> 3` The number of neighbors to spawn a new anticell.

`GOCcellcode = <string> "scalebox{0.03}{drawchicken}"` The  $\text{\LaTeX}$  code for graphical representation of a living cell. You can use basically any valid  $\text{\LaTeX}$  code in here. A chicken is the default, of course.

`GOCanticellcode = <string> "0"` The  $\text{\LaTeX}$  code for graphical representation of a living anticell.

`GOCx = <int> 100` Grid size in x direction (vertical).

`GOCy = <int> 100` Grid size in y direction (horizontal).

`GOCiter = <int> 150` Number of iterations to run the game.

`GOC_console = <bool> false` Activate output on the console.

`GOC_pdf = <bool> true` Activate output in the pdf.

`GOCsleep = <int> 0` Wait after one cycle of the game. This helps especially on the console, or for debugging. By default no wait time is added.

`GOCmakegif = <bool> false` Produce a gif. This requires the command line tool `convert` since I use it for the creation. If you have troubles with this feel free to contact me.

`GOCdensity = <int> 100` Defines the density of the gif export. 100 is quite dense and it might take quite some time to get your gif done.

I recommend to use the `\gameofchicken` with a code roughly like this:

```
\documentclass{scrartcl}
\usepackage{chickenize}
\usepackage[paperwidth=10cm,paperheight=10cm,margin=5mm]{geometry}
\usepackage{graphicx}
\usepackage{placeat}
\placeatsetup{final}
\begin{document}
\gameofchicken{GOCiter=50}
```

```
\gameofchicken{GOCiter=50 GOCmakegif = true}  
  \directlua{ os.execute("gwenview test.gif")} % substitute your filename  
\end{document}
```

Keep in mind that for convenience `\gameofchicken{}` has one argument which is equivalent to using `\chickenizesetup{}` and actually just executes the argument as Lua code ...

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

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

If you use Lua<sub>TeX</sub>, 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 <sub>TeX</sub> behaves: The *callbacks*. A callback is a point where you can hook into <sub>TeX</sub>'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 <sub>TeX</sub>'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.) <sub>TeX</sub> 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 <sub>TeX</sub>'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_filter(head)
  return head
end

callback.register("post_linebreak_filter",my_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 function `luatexbase.add_to_callback`. This is provided by the  $\TeX$  kernel table `luatexbase` which was initially a package by Manuel Pégourié-Gonnard and Élie Roux.<sup>9</sup> This function has a more extended syntax:

```
luatexbase.add_to_callback("post_linebreak_filter",my_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 Lua $\TeX$  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 Lua $\TeX$  manual and the `luatexbase` section in the  $\TeX$  kernel documentation for details!

## 6 Nodes

Essentially everything that Lua $\TeX$  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 Lua $\TeX$  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

---

<sup>9</sup>Since the late 2015 release of  $\TeX$ , the package has not to be loaded anymore since the functionality is absorbed by the kernel. Plain $\TeX$  users can load the `lualatex` file which provides the needed functionality.

the function `node.traverse_id(GLYPH,head)`, with the first argument giving the respective id of the nodes.<sup>10</sup>

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

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

## 7 Other things

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

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

The namespace of the `chickenize` package is *not* consistent. Please don’t take anything here as an example for good Lua coding, for good TeXing or even for good LuaTeXing. It’s not. For high quality code check out the code written by Hans Hagen or other professionals. Once you understand the package at hand, you should be ready to go on and improve your knowledge. After that, you might come back and help me improve this package – I’m always happy for any help ☺

---

<sup>10</sup>GLYPH 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.

## Part III

# Implementation

## 8 T<sub>E</sub>X 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<sub>E</sub>X's attributes.

For (un)registering, we use the luatexbase L<sup>A</sup>T<sub>E</sub>X kernel functionality. Then, the .lua file is loaded which does the actual work. Finally, the T<sub>E</sub>X 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 \directlua{dofile(kpse.find_file("chickenize.lua"))}
2
3 \def\ALT{%
4   \bgroup%
5   \fontspec{Latin Modern Sans}%
6   A%
7   \kern-.375em \raisebox{.65ex}{\scalebox{0.3}{L}}%
8   \kern.03em \raisebox{-.99ex}{T}%
9   \egroup%
10 }
```

### 8.1 allownumberincommands

```
11 \def\allownumberincommands{
12   \catcode`\0=11
13   \catcode`\1=11
14   \catcode`\2=11
15   \catcode`\3=11
16   \catcode`\4=11
17   \catcode`\5=11
18   \catcode`\6=11
19   \catcode`\7=11
20   \catcode`\8=11
21   \catcode`\9=11
22 }
23
24 \def\BEClerialize{
25   \chickenize
26   \directlua{
27     chickenstring[1] = "noise noise"
28     chickenstring[2] = "atom noise"
```



```

29   chickenstring[3] = "shot noise"
30   chickenstring[4] = "photon noise"
31   chickenstring[5] = "camera noise"
32   chickenstring[6] = "noising noise"
33   chickenstring[7] = "thermal noise"
34   chickenstring[8] = "electronic noise"
35   chickenstring[9] = "spin noise"
36   chickenstring[10] = "electron noise"
37   chickenstring[11] = "Bogoliubov noise"
38   chickenstring[12] = "white noise"
39   chickenstring[13] = "brown noise"
40   chickenstring[14] = "pink noise"
41   chickenstring[15] = "bloch sphere"
42   chickenstring[16] = "atom shot noise"
43   chickenstring[17] = "nature physics"
44 }
45 }
46
47 \def\boustrophedon{
48   \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon,"boustrophedon")}}
49 \def\unboustrophedon{
50   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon")}}
51
52 \def\boustrophedonglyphs{
53   \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon_glyphs,"boustrophedonglyphs")}}
54 \def\unboustrophedonglyphs{
55   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon_glyphs")}}
56
57 \def\boustrophedoninverse{
58   \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon_inverse,"boustrophedoninverse")}}
59 \def\unboustrophedoninverse{
60   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon_inverse")}}
61
62 \def\bubblesort{
63   \directlua{luatexbase.add_to_callback("post_linebreak_filter",bubblesort,"bubblesort")}}
64 \def\unbubblesort{
65   \directlua{luatexbase.remove_from_callback("bubblesort","bubblesort")}}
66
67 \def\chickenize{
68   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize")
69   luatexbase.add_to_callback("start_page_number",
70   function() texio.write("[..status.total_pages) end ,"cstartpage")
71   luatexbase.add_to_callback("stop_page_number",
72   function() texio.write(" chickens]") end,"cstoppage")
73   luatexbase.add_to_callback("stop_run",nicetext,"a nice text")
74 }

```

```

75 }
76 \def\unchickenize{
77   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","chickenize")}
78   luatexbase.remove_from_callback("start_page_number","cstartpage")
79   luatexbase.remove_from_callback("stop_page_number","cstoppage")}}
80
81 \def\coffeestainize{ %% to be implemented.
82   \directlua{}}
83 \def\uncoffeestainize{
84   \directlua{}}
85
86 \def\colorstretch{
87   \directlua{luatexbase.add_to_callback("post_linebreak_filter",colorstretch,"stretch_expansion")}
88 \def\uncolorstretch{
89   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","stretch_expansion")}}
90
91 \def\countglyphs{
92   \directlua{
93     counted_glyphs_by_code = {}
94     for i = 1,10000 do
95       counted_glyphs_by_code[i] = 0
96     end
97     glyphnumber = 0 spacenumber = 0
98     luatexbase.add_to_callback("post_linebreak_filter",countglyphs,"countglyphs")
99     luatexbase.add_to_callback("stop_run",printglyphnumber,"printglyphnumber")
100   }
101 }
102
103 \def\countwords{
104   \directlua{wordnumber = 0
105     luatexbase.add_to_callback("pre_linebreak_filter",countwords,"countwords")
106     luatexbase.add_to_callback("stop_run",printwordnumber,"printwordnumber")
107   }
108 }
109
110 \def\detectdoublewords{
111   \directlua{
112     luatexbase.add_to_callback("post_linebreak_filter",detectdoublewords,"detectdoublewords")
113     luatexbase.add_to_callback("stop_run",prindoublewords,"prindoublewords")
114   }
115 }
116
117 \def\dosomethingfunny{
118   %% should execute one of the "funny" commands, but randomly. So every compilation is complete
119   functions. Maybe also on a per-paragraph-basis?
120 }

```

```

120
121 \def\dubstepenize{
122   \chickenize
123   \directlua{
124     chickenstring[1] = "WOB"
125     chickenstring[2] = "WOB"
126     chickenstring[3] = "WOB"
127     chickenstring[4] = "BROOOAR"
128     chickenstring[5] = "WHEE"
129     chickenstring[6] = "WOB WOB WOB"
130     chickenstring[7] = "WAAAAAAAAAH"
131     chickenstring[8] = "duhduh duhduh duh"
132     chickenstring[9] = "BEEEEEEEEEW"
133     chickenstring[10] = "DEEEEEEEEEW"
134     chickenstring[11] = "EEEEEW"
135     chickenstring[12] = "boop"
136     chickenstring[13] = "buhdee"
137     chickenstring[14] = "bee bee"
138     chickenstring[15] = "BZZRRRRRRRROOOOOOAAAAA"
139
140     chickenizefraction = 1
141   }
142 }
143 \let\dubstepize\dubstepenize
144
145 \def\explainbackslashes{ %% inspired by xkcd #1638
146   {\tt\noindent
147   \textbackslash escape character\
148   \textbackslash\textbackslash line end or escaped escape character in tex.print("")\
149   \textbackslash\textbackslash\textbackslash real, real backslash\
150   \textbackslash\textbackslash\textbackslash\textbackslash line end in tex.print("")\
151   \textbackslash\textbackslash\textbackslash\textbackslash\textbackslash elder backslash \
152   \textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash backslash wh
153   \textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash
154   \textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash
155   \textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash
   eater}
156 }
157
158 \def\francize{
159   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",francize,"francize")}}
160
161 \def\unfrancize{
162   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter",francize)}}
163
164 \def\gameoflife{

```

```

165 Your Life Is Tetris. Stop Playing It Like Chess.
166 }

```

This is just the activation of the command, the typesetting is done in the Lua code/loop as explained below. Use this macro *after* `\begin{document}`. Remember that `graphicx` and `placeat` are required!

```

167 \def\gameofchicken#1{\directlua{
168   GOCrule_live = {2,3}
169   GOCrule_spawn = {3}
170   GOCrule_antilive = {2,3}
171   GOCrule_antispawn = {3}
172   GOCcellcode = "\\scalebox{0.03}{\\drawchicken}"
173   GOCcellcode = "\\scalebox{0.03}{\\drawcov}"
174   GOCx = 100
175   GOCy = 100
176   GOCiter = 150
177   GOC_console = false
178   GOC_pdf = true
179   GOCsleep = 0
180   GOCdensity = 100
181   #1
182   gameofchicken()
183
184   if (GOCmakegif == true) then
185     luatexbase.add_to_callback("wrapup_run",make_a_gif,"makeagif")
186   end
187 }}
188 \let\gameofchimken\gameofchicken % yeah, that had to be.
189
190 \def\gutenbergenize{ %% makes only sense when using LaTeX
191   \AtBeginDocument{
192     \let\grqq\relax\let\glqq\relax
193     \let\frqq\relax\let\flqq\relax
194     \let\grq\relax\let\glq\relax
195     \let\frq\relax\let\flq\relax
196   }
197   \gdef\footnote##1{}
198   \gdef\cite##1{}\gdef\parencite##1{}
199   \gdef\Cite##1{}\gdef\Parencite##1{}
200   \gdef\cites##1{}\gdef\parencites##1{}
201   \gdef\Cites##1{}\gdef\Parencites##1{}
202   \gdef\footcite##1{}\gdef\footcitetext##1{}
203   \gdef\footcites##1{}\gdef\footcitetexts##1{}
204   \gdef\textcite##1{}\gdef\Textcite##1{}
205   \gdef\textcites##1{}\gdef\Textcites##1{}
206   \gdef\smartcites##1{}\gdef\Smartcites##1{}
207   \gdef\supercite##1{}\gdef\supercites##1{}
208   \gdef\autocite##1{}\gdef\Autocite##1{}

```

```

209 \gdef\autocites##1{}\gdef\Autocites##1{}
210 %% many, many missing ... maybe we need to tackle the underlying mechanism?
211 }
212 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",guttenbergenize_rq,"guttenbergenize_rq")}
213 }
214
215 \def\hammertime{
216 \global\let\n\relax
217 \directlua{hammerfirst = true
218 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",hammertime,"hammertime")}}
219 \def\unhammertime{
220 \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","hammertime")}}
221
222 \let\hendlnize\chickenize % homage to Hendl/Chicken
223 \let\unhendlnize\unchickenize % may the soldering strength always be with him
224
225 \def\italianizerandword{
226 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",italianizerandword,"italianizerandword")}
227 \def\unitalianizerandword{
228 \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","italianizerandword")}}
229
230 \def\italianize{
231 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",italianize,"italianize")}}
232 \def\unitalianize{
233 \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","italianize")}}
234
235 % \def\itsame{
236 % \directlua{drawmario}} %%% does not exist
237
238 \def\kernmanipulate{
239 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",kernmanipulate,"kernmanipulate")}}
240 \def\unkernmanipulate{
241 \directlua{luatexbase.remove_from_callback("pre_linebreak_filter",kernmanipulate)}}
242
243 \def\leetspeak{
244 \directlua{luatexbase.add_to_callback("post_linebreak_filter",leet,"1337")}}
245 \def\unleetspeak{
246 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","1337")}}
247
248 \def\leftsideright#1{
249 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",leftsideright,"leftsideright")}}
250 \directlua{
251 leftsiderightindex = {#1}
252 leftsiderightarray = {}
253 for _,i in pairs(leftsiderightindex) do
254 leftsiderightarray[i] = true

```

```

255     end
256 }
257 }
258 \def\unleftsideright{
259   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","leftsideright")}}
260
261 \def\letterspaceadjust{
262   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",letterspaceadjust,"letterspaceadjust")}}
263 \def\unletterspaceadjust{
264   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","letterspaceadjust")}}
265
266 \def\listallcommands{
267   \directlua{
268     for name in pairs(tex.hashtokens()) do
269       print(name)
270     end}
271 }
272
273 \let\stealsheep\letterspaceadjust      %% synonym in honor of Paul
274 \let\unstealsheep\unletterspaceadjust
275 \let\returnsheep\unletterspaceadjust
276
277 \def\matrixize{
278   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",matrixize,"matrixize")}}
279 \def\unmatrixize{
280   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","matrixize")}}
281
282 \def\milkcow{      %% FIXME %% to be implemented
283   \directlua{}}
284 \def\unmilkcow{
285   \directlua{}}
286
287 \def\medievalumlaut{
288   \directlua{luatexbase.add_to_callback("post_linebreak_filter",medievalumlaut,"medievalumlaut")}}
289 \def\unmedievalumlaut{
290   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","medievalumlaut")}}
291
292 \def\pancakenize{
293   \directlua{luatexbase.add_to_callback("stop_run",pancaketext,"pancaketext")}}
294
295 \def\rainbowcolor{
296   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"rainbowcolor")
297     rainbowcolor = true}}
298 \def\unrainbowcolor{
299   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","rainbowcolor")
300     rainbowcolor = false}}

```

```

301 \let\nyanize\rainbowcolor
302 \let\unnyanize\unrainbowcolor
303
304 \def\randomchars{
305   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomchars,"randomchars")}}
306 \def\unrandomchars{
307   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomchars")}}
308
309 \def\randomcolor{
310   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"randomcolor")}}
311 \def\unrandomcolor{
312   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomcolor")}}
313
314 \def\randomerror{ %% FIXME
315   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomerror,"randomerror")}}
316 \def\unrandomerror{ %% FIXME
317   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomerror")}}
318
319 \def\randomfonts{
320   \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomfonts,"randomfonts")}}
321 \def\unrandomfonts{
322   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomfonts")}}
323
324 \def\randomuclc{
325   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",randomuclc,"randomuclc")}}
326 \def\unrandomuclc{
327   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","randomuclc")}}
328
329 \def\relationship{%
330   \directlua{luatexbase.add_to_callback("post_linebreak_filter",cutparagraph,"cut paragraph")
331     relationship()
332   }
333 }
334
335 \let\rongorongonize\boustrophedoninverse
336 \let\unrongorongonize\unboustrophedoninverse
337
338 \def\scorpionize{
339   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",scorpionize_color,"scorpionize_color")}}
340 \def\unscorpionize{
341   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","scorpionize_color")}}
342
343 \def\spankmonkey{ %% to be implemented
344   \directlua{}}
345 \def\unspankmonkey{
346   \directlua{}}

```

```

347
348 \def\substitutewords{
349   \directlua{luatexbase.add_to_callback("process_input_buffer",substitutewords,"substitutewords")}
350 \def\unsubstitutewords{
351   \directlua{luatexbase.remove_from_callback("process_input_buffer","substitutewords")}}
352
353 \def\addtosubstitutions#1#2{
354   \directlua{addtosubstitutions("#1","#2")}}
355 }
356
357 \def\suppressonecharbreak{
358   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",suppressonecharbreak,"suppressonecharbreak")}
359 \def\unsuppressonecharbreak{
360   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","suppressonecharbreak")}}
361
362 \def\tabularasa{
363   \directlua{luatexbase.add_to_callback("post_linebreak_filter",tabularasa,"tabularasa")}}
364 \def\untabularasa{
365   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","tabularasa")}}
366
367 \def\tanjanize{
368   \directlua{luatexbase.add_to_callback("post_linebreak_filter",tanjanize,"tanjanize")}}
369 \def\untanjanize{
370   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","tanjanize")}}
371
372 \def\uppercasecolor{
373   \directlua{luatexbase.add_to_callback("post_linebreak_filter",uppercasecolor,"uppercasecolor")}}
374 \def\unuppercasecolor{
375   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","uppercasecolor")}}
376
377 \def\upsideown#1{
378   \directlua{luatexbase.add_to_callback("post_linebreak_filter",upsideown,"upsideown")}}
379   \directlua{
380     upsideownindex = {#1}
381     upsideownarray = {}
382     for _,i in pairs(upsideownindex) do
383       upsideownarray[i] = true
384     end
385   }
386 }
387 \def\unupsideown{
388   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","upsideown")}}
389
390 \def\variantjustification{
391   \directlua{luatexbase.add_to_callback("post_linebreak_filter",variantjustification,"variantjustification")}}
392 \def\unvariantjustification{

```



```

393 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","variantjustification")}}
394
395 \def\zebranize{
396 \directlua{luatexbase.add_to_callback("post_linebreak_filter",zebranize,"zebranize")}}
397 \def\unzebranize{
398 \directlua{luatexbase.remove_from_callback("post_linebreak_filter","zebranize")}}

```

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

```

399 \newattribute\leetattr
400 \newattribute\letterspaceadjustattr
401 \newattribute\randcolorattr
402 \newattribute\randfontssattr
403 \newattribute\randuclcatr
404 \newattribute\tabularasaattr
405 \newattribute\uppercasecolorattr
406
407 \long\def\textleetspeak#1%
408 {\setluatexattribute\leetattr{42}#1\unsetluatexattribute\leetattr}
409
410 \long\def\textletterspaceadjust#1{
411 \setluatexattribute\letterspaceadjustattr{42}#1\unsetluatexattribute\letterspaceadjustattr
412 \directlua{
413   if (textletterspaceadjustactive) then else % -- if already active, do nothing
414     luatexbase.add_to_callback("pre_linebreak_filter",textletterspaceadjust,"textletterspaceadjust")
415   end
416   textletterspaceadjustactive = true           % -- set to active
417 }
418 }
419 \let\textlsa\textletterspaceadjust
420
421 \long\def\textrandomcolor#1%
422 {\setluatexattribute\randcolorattr{42}#1\unsetluatexattribute\randcolorattr}
423 \long\def\textrandomfontss#1%
424 {\setluatexattribute\randfontssattr{42}#1\unsetluatexattribute\randfontssattr}
425 \long\def\textrandomfontss#1%
426 {\setluatexattribute\randfontssattr{42}#1\unsetluatexattribute\randfontssattr}
427 \long\def\textrandomuclc#1%
428 {\setluatexattribute\randuclcatr{42}#1\unsetluatexattribute\randuclcatr}
429 \long\def\texttabularasa#1%
430 {\setluatexattribute\tabularasaattr{42}#1\unsetluatexattribute\tabularasaattr}
431 \long\def\textuppercasecolor#1%
432 {\setluatexattribute\uppercasecolorattr{42}#1\unsetluatexattribute\uppercasecolorattr}

```

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

```

433 \def\chickenizesetup#1{\directlua{#1}}

```

## 8.2 drawchicken

The following is the very first try of implementing a small drawing language in Lua. It draws a beautiful (?) chicken. TODO: Make it scalable by giving relative sizes. Also: Allow it to look to the other side if wanted.

```
434 \long\def\luadraw#1#2{%
435   \vbox to #1bp{%
436     \vfil
437     \latelua{pdf_print("q") #2 pdf_print("Q")}%
438   }%
439 }
440 \long\def\drawchicken{
441   \luadraw{90}{
442     chickenhead      = {200,50} % chicken head center
443     chickenhead_rad = 20
444
445     neckstart = {215,35} % neck
446     neckstop  = {230,10} %
447
448     chickenbody      = {260,-10}
449     chickenbody_rad = 40
450     chickenleg = {
451       {{260,-50},{250,-70},{235,-70}},
452       {{270,-50},{260,-75},{245,-75}}
453     }
454
455     beak_top = {185,55}
456     beak_front = {165,45}
457     beak_bottom = {185,35}
458
459     wing_front = {260,-10}
460     wing_bottom = {280,-40}
461     wing_back = {275,-15}
462
463     sloppycircle(chickenhead,chickenhead_rad) sloppylines(neckstart,neckstop)
464     sloppycircle(chickenbody,chickenbody_rad)
465     sloppylines(chickenleg[1][1],chickenleg[1][2]) sloppylines(chickenleg[1][2],chickenleg[1][3])
466     sloppylines(chickenleg[2][1],chickenleg[2][2]) sloppylines(chickenleg[2][2],chickenleg[2][3])
467     sloppylines(beak_front,beak_top) sloppylines(beak_front,beak_bottom)
468     sloppylines(wing_front,wing_bottom) sloppylines(wing_back,wing_bottom)
469   }
470 }
```

## 8.3 drawcov

This draws a corona virus since I had some time to work on this package due to the shutdown caused by COVID-19.

```

471 \long\def\drawcov{
472   \luadraw{90}{
473     covbody = {200,50}
474     covbody_rad = 50
475
476     covcrown_rad = 5
477     crownno = 13
478     for i=1,crownno do
479       crownpos = {covbody[1]+1.4*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+1.4*covbody_rad}
480       crownconnect = {covbody[1]+covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+covbody_rad}
481       sloppycircle(crownpos,covcrown_rad)
482       sloppyline(crownpos,crownconnect)
483     end
484
485     covcrown_rad = 6
486     crownno = 8
487     for i=1,crownno do
488       crownpos = {covbody[1]+0.8*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+0.8*covbody_rad}
489       crownconnect = {covbody[1]+0.5*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+0.5*covbody_rad}
490       sloppycircle(crownpos,covcrown_rad)
491       sloppyline(crownpos,crownconnect)
492     end
493
494     covcrown_rad = 8
495     sloppycircle(covbody,covcrown_rad)
496     sloppycircle(covbody,covbody_rad)
497     sloppyline(covbody,covbody)
498   }
499 }

```

## 9 L<sup>A</sup>T<sub>E</sub>X package

I have decided to keep the L<sup>A</sup>T<sub>E</sub>X-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.

```

500 \ProvidesPackage{chickenize}%
501 [2020/04/20 v0.2.9 chickenize package]
502 \input{chickenize}

```

### 9.1 Free Compliments

503 %

## 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! :)

```
504 \iffalse
505 \DeclareDocumentCommand\includegraphics{O{m}}{
506     \fbox{Chicken} %% actually, I'd love to draw an MP graph showing a chicken ...
507 }
508 %%% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
509 %% So far, you have to load pgfplots yourself.
510 %% As it is a mighty package, I don't want the user to force loading it.
511 \NewDocumentCommand\balmerpeak{G{}O{-4cm}}{
512 %% to be done using Lua drawing.
513 }
514 \fi
```

## 10 Lua Module

This file contains all the necessary functions and is the actual work horse of this package. The functions are sorted 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.

```
515
516 local nodeid    = node.id
517 local nodecopy  = node.copy
518 local nodenew   = node.new
519 local nodetail  = node.tail
520 local nodeslide = node.slide
521 local noderemove = node.remove
522 local nodetraverseid = node.traverse_id
523 local nodeinsertafter = node.insert_after
524 local nodeinsertbefore = node.insert_before
525
526 Hhead = nodeid("hhead")
527 RULE  = nodeid("rule")
528 GLUE  = nodeid("glue")
529 WHAT  = nodeid("whatsit")
530 COL   = node.subtype("pdf_colorstack")
531 DISC  = nodeid("disc")
532 GLYPH = nodeid("glyph")
533 GLUE  = nodeid("glue")
534 HLIST = nodeid("hlist")
535 KERN  = nodeid("kern")
536 PUNCT = nodeid("punct")
537 PENALTY = nodeid("penalty")
```

```
538 PDF_LITERAL = node.subtype("pdf_literal")
```

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

```
539 color_push = nodenew(WHAT,COL)
```

```
540 color_pop = nodenew(WHAT,COL)
```

```
541 color_push.stack = 0
```

```
542 color_pop.stack = 0
```

```
543 color_push.command = 1
```

```
544 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.

```
545 chicken_pagenumbers = true
```

```
546
```

```
547 chickenstring = {}
```

```
548 chickenstring[1] = "chicken" -- chickenstring is a table, please remeber this!
```

```
549
```

```
550 chickenizefraction = 0.5 -- set this to a small value to fool somebody, or to see if your text has
```

```
551 chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing your
```

```
552
```

```
553 local match = unicode.utf8.match
```

```
554 chickenize_ignore_word = false
```

The function chickenize\_real\_stuff is started once the beginning of a to-be-substituted word is found.

```
555 chickenize_real_stuff = function(i,head)
```

```
556   while ((i.next.id == GLYPH) or (i.next.id == KERN) or (i.next.id == DISC) or (i.next.id == HL) or (i.next.id == VERT) or (i.next.id == HIDE))
557     find end of a word
```

```
557     i.next = i.next.next
```

```
558   end
```

```
559
```

```
560   chicken = {} -- constructing the node list.
```

```
561
```

```
562 -- Should this be done only once? No, otherwise we lose the freedom to change the string in-
563 -- document.
```

```
564 -- But it could be done only once each paragraph as in-paragraph changes are not possible!
```

```
565
```

```
565   chickenstring_tmp = chickenstring[math.random(1,#chickenstring)]
```

```
566   chicken[0] = nodenew(GLYPH,1) -- only a dummy for the loop
```

```
567   for i = 1,string.len(chickenstring_tmp) do
```

```
568     chicken[i] = nodenew(GLYPH,1)
```

```
569     chicken[i].font = font.current()
```

```
570     chicken[i-1].next = chicken[i]
```

```
571   end
```

```
572
```

```

573     j = 1
574     for s in string.utfvalues(chickenstring_tmp) do
575         local char = unicode.utf8.char(s)
576         chicken[j].char = s
577         if match(char,"%s") then
578             chicken[j] = nodenew(GLUE)
579             chicken[j].width = space
580             chicken[j].shrink = shrink
581             chicken[j].stretch = stretch
582         end
583         j = j+1
584     end
585
586     nodeslide(chicken[1])
587     lang.hyphenate(chicken[1])
588     chicken[1] = node.kerning(chicken[1])    -- FIXME: does not work
589     chicken[1] = node.ligaturing(chicken[1]) -- dito
590
591     nodeinsertbefore(head,i,chicken[1])
592     chicken[1].next = chicken[2] -- seems to be necessary ... to be fixed
593     chicken[string.len(chickenstring_tmp)].next = i.next
594
595     -- shift lowercase latin letter to uppercase if the original input was an uppercase
596     if (chickenize_capital and (chicken[1].char > 96 and chicken[1].char < 123)) then
597         chicken[1].char = chicken[1].char - 32
598     end
599
600     return head
601 end
602
603 chickenize = function(head)
604     for i in nodetraverseid(GLYPH,head) do --find start of a word
605         -- Random determination of the chickenization of the next word:
606         if math.random() > chickenizefraction then
607             chickenize_ignore_word = true
608         elseif chickencount then
609             chicken_substitutions = chicken_substitutions + 1
610         end
611
612         if (chickenize_ignore_word == false) then -- normal case: at the beginning of a word, we jump
613             if (i.char > 64 and i.char < 91) then chickenize_capital = true else chickenize_capital = false
614             head = chickenize_real_stuff(i,head)
615         end
616
617 -- At the end of the word, the ignoring is reset. New chance for everyone.
618         if not((i.next.id == GLYPH) or (i.next.id == DISC) or (i.next.id == PUNCT) or (i.next.id == KERN)) then

```

```

619     chickenize_ignore_word = false
620   end
621 end
622 return head
623 end
624

```

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)

```

625 local separator      = string.rep("=", 28)
626 local texiowrite_nl = texio.write_nl
627 nicetext = function()
628   texiowrite_nl("Output written on "..tex.jobname..".pdf ("..status.total_pages.." chicken,".." e
629   texiowrite_nl(" ")
630   texiowrite_nl(separator)
631   texiowrite_nl("Hello my dear user,")
632   texiowrite_nl("good job, now go outside and enjoy the world!")
633   texiowrite_nl(" ")
634   texiowrite_nl("And don't forget to feed your chicken!")
635   texiowrite_nl(separator .. "\n")
636   if chickencount then
637     texiowrite_nl("There were "..chicken_substitutions.." substitutions made.")
638     texiowrite_nl(separator)
639   end
640 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:

```

641 boustrophedon = function(head)
642   rot = node.new(WHAT,PDF_LITERAL)
643   rot2 = node.new(WHAT,PDF_LITERAL)
644   odd = true
645   for line in node.traverse_id(0,head) do
646     if odd == false then
647       w = line.width/65536*0.99625 -- empirical correction factor (?)
648       rot.data = "-1 0 0 1 "..w.." 0 cm"
649       rot2.data = "-1 0 0 1 "..-w.." 0 cm"
650       line.head = node.insert_before(line.head,line.head,nodecopy(rot))
651       nodeinsertafter(line.head,nodetail(line.head),nodecopy(rot2))
652       odd = true
653     else
654       odd = false
655     end

```

```

656     end
657   return head
658 end

```

Glyphwise rotation:

```

659 boustrophedon_glyphs = function(head)
660   odd = false
661   rot = nodenew(WHAT,PDF_LITERAL)
662   rot2 = nodenew(WHAT,PDF_LITERAL)
663   for line in nodetraverseid(0,head) do
664     if odd==true then
665       line.dir = "TRT"
666       for g in nodetraverseid(GLYPH,line.head) do
667         w = -g.width/65536*0.99625
668         rot.data = "-1 0 0 1 " .. w .. " 0 cm"
669         rot2.data = "-1 0 0 1 " .. -w .. " 0 cm"
670         line.head = node.insert_before(line.head,g,nodecopy(rot))
671         nodeinsertafter(line.head,g,nodecopy(rot2))
672       end
673       odd = false
674     else
675       line.dir = "TLT"
676       odd = true
677     end
678   end
679   return head
680 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.

```

681 boustrophedon_inverse = function(head)
682   rot = node.new(WHAT,PDF_LITERAL)
683   rot2 = node.new(WHAT,PDF_LITERAL)
684   odd = true
685   for line in node.traverse_id(0,head) do
686     if odd == false then
687 texio.write_nl(line.height)
688       w = line.width/65536*0.99625 -- empirical correction factor (?)
689       h = line.height/65536*0.99625
690       rot.data = "-1 0 0 -1 "..w.." "..h.." cm"
691       rot2.data = "-1 0 0 -1"..-w.." "..0.5*h.." cm"
692       line.head = node.insert_before(line.head,line.head,node.copy(rot))
693       node.insert_after(line.head,node.tail(line.head),node.copy(rot2))
694       odd = true
695     else
696       odd = false
697     end

```



```

698     end
699   return head
700 end

```

### 10.3 bubblesort

Bubblesort is to be implemented. Why? Because it's funny.

```

701 function bubblesort(head)
702   for line in nodetraverseid(0,head) do
703     for glyph in nodetraverseid(GLYPH,line.head) do
704
705     end
706   end
707   return head
708 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!

```

709 countglyphs = function(head)
710   for line in nodetraverseid(0,head) do
711     for glyph in nodetraverseid(GLYPH,line.head) do
712       glyphnumber = glyphnumber + 1
713       if (glyph.next.next) then
714         if (glyph.next.id == 10) and (glyph.next.next.id == GLYPH) then
715           spacenumber = spacenumber + 1
716         end
717         counted_glyphs_by_code[glyph.char] = counted_glyphs_by_code[glyph.char] + 1
718       end
719     end
720   end
721   return head
722 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? ...

```

723 printglyphnumber = function()
724   texiowrite_nl("\nNumber of glyphs by character code (only up to 127):")
725   for i = 1,127 do --%% FIXME: should allow for more characters, but cannot be printed to console
726     texiowrite_nl(string.char(i)..": " ..counted_glyphs_by_code[i])
727   end
728
729   texiowrite_nl("\nTotal number of glyphs in this document: " ..glyphnumber)
730   texiowrite_nl("Number of spaces in this document: " ..spacenumber)
731   texiowrite_nl("Glyphs plus spaces: " ..glyphnumber+spacenumber.." \n")
732 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.

```

733 countwords = function(head)
734   for glyph in nodetraverseid(GLYPH,head) do
735     if (glyph.next.id == 10) then
736       wordnumber = wordnumber + 1
737     end
738   end
739   wordnumber = wordnumber + 1 -- add 1 for the last word in a paragraph which is not found otherwise
740   return head
741 end

```

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

```

742 printwordnumber = function()
743   texiowrite_nl("\nNumber of words in this document: " ..wordnumber)
744 end

```

## 10.6 detectdoublewords

```

745 %% FIXME: Does this work? ...
746 detectdoublewords = function(head)
747   prevlastword = {} -- array of numbers representing the glyphs
748   prevfirstword = {}
749   newlastword = {}
750   newfirstword = {}
751   for line in nodetraverseid(0,head) do
752     for g in nodetraverseid(GLYPH,line.head) do
753       texio.write_nl("next glyph",#newfirstword+1)

```

```

754     newfirstword[#newfirstword+1] = g.char
755     if (g.next.id == 10) then break end
756 end
757 texio.write_nl("nfw: "..#newfirstword)
758 end
759 end
760
761 printdoublewords = function()
762   texio.write_nl("finished")
763 end

```

## 10.7 francize

This function is intentionally undocumented. It randomizes all numbers digit by digit. Why? Because.

```

764 francize = function(head)
765   for n in nodetraverseid(GLYPH,head) do
766     if ((n.char > 47) and (n.char < 58)) then
767       n.char = math.random(48,57)
768     end
769   end
770   return head
771 end

```

## 10.8 gamofchicken

The `gameofchicken` is an implementation of the Game of Life by Conway. The standard cell here is a chicken, while there are also anticells. For both you can adapt the  $\text{\LaTeX}$  code to represent the cells.

I also kick in some code to convert the pdf into a gif after the pdf has been finalized and  $\text{\LaTeX}$  is about to end. This uses a system call to `convert`; especially the latter one will change. For now this is a convenient implementation for me and maybe most Linux environments to get the gif by one-click-compiling the `tex` document.

```

772 function gameofchicken()
773   GOC_lifetab = {}
774   GOC_spawnab = {}
775   GOC_antilifetab = {}
776   GOC_antispawnab = {}
777   -- translate the rules into an easily-manageable table
778   for i=1,#GOCrule_live do; GOC_lifetab[GOCrule_live[i]] = true end
779   for i=1,#GOCrule_spawn do; GOC_spawnab[GOCrule_spawn[i]] = true end
780   for i=1,#GOCrule_antilive do; GOC_antilifetab[GOCrule_antilive[i]] = true end
781   for i=1,#GOCrule_antispawn do; GOC_antispawnab[GOCrule_antispawn[i]] = true end

```

Initialize the arrays for cells and anticells with zeros.

```

782 -- initialize the arrays
783 local life = {}
784 local antilife = {}
785 local newlife = {}

```

```

786 local newantilife = {}
787 for i = 0, GOCx do life[i] = {}; newlife[i] = {} for j = 0, GOCy do life[i][j] = 0 end end
788 for i = 0, GOCx do antilife[i] = {}; newantilife[i] = {} for j = 0, GOCy do antilife[i][j] = 0 end end

```

These are the functions doing the actual work, checking the neighbors and applying the rules defined above.

```

789 function applyruleslife(neighbors, lifeij, antineighbors, antilifeij)
790   if GOC_spawntab[neighbors] then myret = 1 else -- new cell
791     if GOC_lifetab[neighbors] and (lifeij == 1) then myret = 1 else myret = 0 end end
792   if antineighbors > 1 then myret = 0 end
793   return myret
794 end
795 function applyrulesantilife(neighbors, lifeij, antineighbors, antilifeij)
796   if (antineighbors == 3) then myret = 1 else -- new cell or keep cell
797     if (((antineighbors > 1) and (antineighbors < 4)) and (lifeij == 1)) then myret = 1 else myret = 0 end
798   if neighbors > 1 then myret = 0 end
799   return myret
800 end

```

Preparing the initial state with a default pattern:

```

801 -- prepare some special patterns as starter
802 life[53][26] = 1 life[53][25] = 1 life[54][25] = 1 life[55][25] = 1 life[54][24] = 1

```

And the main loop running from here:

```

803 print("start");
804 for i = 1, GOCx do
805   for j = 1, GOCy do
806     if (life[i][j] == 1) then texio.write("X") else if (antilife[i][j] == 1) then texio.write("0") end
807   end
808   texio.write_nl(" ");
809 end
810 os.sleep(GOCsleep)
811
812 for i = 0, GOCx do
813   for j = 0, GOCy do
814     newlife[i][j] = 0 -- Fill the values from the start settings here
815     newantilife[i][j] = 0 -- Fill the values from the start settings here
816   end
817 end
818
819 for k = 1, GOCiter do -- iterate over the cycles
820   texio.write_nl(k);
821   for i = 1, GOCx-1 do -- iterate over lines
822     for j = 1, GOCy-1 do -- iterate over columns -- prevent edge effects
823       local neighbors = (life[i-1][j-1] + life[i-1][j] + life[i-1][j+1] + life[i][j-1] +
824         life[i][j+1] + life[i+1][j-1] + life[i+1][j] + life[i+1][j+1])
825       local antineighbors = (antilife[i-1][j-1] + antilife[i-1][j] + antilife[i-1][j+1] +
826         antilife[i][j-1] + antilife[i][j+1] + antilife[i+1][j-1] + antilife[i+1][j] + antilife[i+1][j+1])

```

```

826         newlife[i][j] = applyruleslife(neighbors, life[i][j],antineighbors, antilife[i][j])
827         newantilife[i][j] = applyrulesantilife(neighbors,life[i][j], antineighbors,antilife[i][j])
828     end
829 end
830
831 for i = 1, GOCx do
832     for j = 1, GOCy do
833         life[i][j] = newlife[i][j] -- copy the values
834         antilife[i][j] = newantilife[i][j] -- copy the values
835     end
836 end
837
838 for i = 1,GOCx do
839     for j = 1,GOCy do
840         if GOC_console then
841             if (life[i][j]==1) then texio.write("X") else if (antilife[i][j]==1) then texio.write("O")
842         end
843         if GOC_pdf then
844             if (life[i][j]==1) then tex.print("\\placeat("..(i/10)..","..(j/10).."){"..GOCcellcode..
845             if (antilife[i][j]==1) then tex.print("\\placeat("..(i/10)..","..(j/10).."){"..GOCanticode..
846         end
847     end
848 end
849 tex.print(".\\newpage")
850 os.sleep(GOCsleep)
851 end
852 end --end function gameofchicken

```

The following is a function calling some tool from your operating system. This requires of course that you have them present – that should be the case on a typical Linux distribution. Take care that convert normally does not allow for conversion from pdf, please check that this is allowed by the rules. So this is more an example code that can help you to add it to your game so you can enjoy your chickens developing as a gif.

```

853 function make_a_gif()
854     os.execute("convert -verbose -dispose previous -background white -alpha remove -
      alpha off -density "..GOCdensity.." "..tex.jobname ..".pdf " ..tex.jobname..".gif")
855     os.execute("gwenview "..tex.jobname..".gif")
856 end

```

## 10.9 guttenbergenize

A function in honor of the German politician Gutenberg.<sup>11</sup> Please do *not* confuse him with the grand master Gutenberg!

Calling `\gutenbergenize` will not only execute or manipulate Lua code, but also redefine some  $\TeX$  or  $\LaTeX$  commands. The aim is to remove all quotations, footnotes and anything that will give information

---

<sup>11</sup>Thanks to Jasper for bringing me to this idea!

about the real sources of your work.

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

### 10.9.1 guttenbergenize – preliminaries

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

```
857 local quotestrings = {
858     [171] = true, [172] = true,
859     [8216] = true, [8217] = true, [8218] = true,
860     [8219] = true, [8220] = true, [8221] = true,
861     [8222] = true, [8223] = true,
862     [8248] = true, [8249] = true, [8250] = true,
863 }
```

### 10.9.2 guttenbergenize – the function

```
864 guttenbergenize_rq = function(head)
865     for n in nodetraverseid(GLYPH, head) do
866         local i = n.char
867         if quotestrings[i] then
868             noderemove(head, n)
869         end
870     end
871     return head
872 end
```

### 10.10 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.<sup>12</sup>

```
873 hammertimedelay = 1.2
874 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
875 hammertime = function(head)
876     if hammerfirst then
877         texiowrite_nl(htime_separator)
878         texiowrite_nl("=====STOP!=====\\n")
879         texiowrite_nl(htime_separator .. "\\n\\n\\n")
880         os.sleep (hammertimedelay*1.5)
881         texiowrite_nl(htime_separator .. "\\n")
882         texiowrite_nl("=====HAMMERTIME=====\\n")
883         texiowrite_nl(htime_separator .. "\\n\\n")
884         os.sleep (hammertimedelay)
```

---

<sup>12</sup><http://tug.org/pipermail/luatex/2011-November/003355.html>

```

885     hammerfirst = false
886   else
887     os.sleep (hammertimedelay)
888     texiowrite_nl(htime_separator)
889     texiowrite_nl("====U can't touch this!====\n")
890     texiowrite_nl(htime_separator .. "\n\n")
891     os.sleep (hammertimedelay*0.5)
892   end
893   return head
894 end

```

## 10.11 italianize

This is inspired by some of the more melodic pronunciations of the english language. The command will add randomly an h in front of every word starting with a vowel or remove h from words starting with one. Also, it will add randomly an e to words ending in consonants. This is tricky and might fail – I’m happy to receive and try to solve any bug reports.

```

895 italianizefraction = 0.5 --%% gives the amount of italianization
896 mynode = nodenew(GLYPH) -- prepare a dummy glyph
897
898 italianize = function(head)
899   -- skip "h/H" randomly
900   for n in node.traverse_id(GLYPH,head) do -- go through all glyphs
901     if n.prev.id ~= GLYPH then -- check if it's a word start
902       if ((n.char == 72) or (n.char == 104)) and (tex.normal_rand() < italianizefraction) then --
903         n.prev.next = n.next
904       end
905     end
906   end
907
908   -- add h or H in front of vowels
909   for n in node.traverse_id(GLYPH,head) do
910     if math.random() < italianizefraction then
911       x = n.char
912       if x == 97 or x == 101 or x == 105 or x == 111 or x == 117 or
913         x == 65 or x == 69 or x == 73 or x == 79 or x == 85 then
914         if (n.prev.id == GLUE) then
915           mynode.font = n.font
916           if x > 90 then -- lower case
917             mynode.char = 104
918           else
919             mynode.char = 72 -- upper case - convert into lower case
920             n.char = x + 32
921           end
922           node.insert_before(head,n,node.copy(mynode))
923         end

```

```

924     end
925   end
926 end
927
928 -- add e after words, but only after consonants
929 for n in node.traverse_id(GLUE,head) do
930   if n.prev.id == GLYPH then
931     x = n.prev.char
932     -- skip vowels and randomize
933     if not(x == 97 or x == 101 or x == 105 or x == 111 or x == 117 or x == 44 or x == 46) and matl
934       mynode.char = 101          -- it's always a lower case e, no?
935       mynode.font = n.prev.font -- adapt the current font
936       node.insert_before(head,n,node.copy(mynode)) -- insert the e in the node list
937     end
938   end
939 end
940
941 return head
942 end
943 % \subsection{italianize}\label{sec:italianizerandword}
944 % This is inspired by my dearest colleagues and their artistic interpretation of the english gram
945 % \begin{macrocode}
946 italianizerandwords = function(head)
947 words = {}
948 -- head.next.next is the very first word. However, let's try to get the first word after the first
949 wordnumber = 0
950 for n in node.traverseid(GLUE,head) do -- let's try to count words by their separators
951   wordnumber = wordnumber + 1
952   if n.next then
953     texio.write_nl(n.next.char)
954     words[wordnumber] = {}
955     words[wordnumber][1] = node.copy(n.next)
956
957     glyphnumber = 1
958     myglyph = n.next
959     while myglyph.next do
960       node.tail(words[wordnumber][1]).next = node.copy(myglyph.next)
961       myglyph = myglyph.next
962     end
963   end
964 end
965 myinsertnode = head.next.next -- first letter
966 node.tail(words[1][1]).next = myinsertnode.next
967 myinsertnode.next = words[1][1]
968
969 return head

```



```

970 end
971
972 italianize_old = function(head)
973   local wordlist = {} -- here we will store the number of words of the sentence.
974   local words = {} -- here we will store the words of the sentence.
975   local wordnumber = 0
976   -- let's first count all words in one sentence, howboutdat?
977   wordlist[wordnumber] = 1 -- let's save the word *length* in here ...
978
979
980   for n in nodetraverseid(GLYPH,head) do
981     if (n.next.id == GLUE) then -- this is a space
982       wordnumber = wordnumber + 1
983       wordlist[wordnumber] = 1
984       words[wordnumber] = n.next.next
985     end
986     if (n.next.id == GLYPH) then -- it's a glyph
987       if (n.next.char == 46) then -- this is a full stop.
988         wordnumber = wordnumber + 1
989         texio.write_nl("this sentence had "..wordnumber.."words.")
990         for i=0,wordnumber-1 do
991           texio.write_nl("word "..i.." had " .. wordlist[i] .. "glyphs")
992         end
993         texio.write_nl(" ")
994         wordnumber = -1 -- to compensate the fact that the next node will be a space, this would co
995       else
996
997         wordlist[wordnumber] = wordlist[wordnumber] + 1 -- the current word got 1 glyph longer
998       end
999     end
1000   end
1001   return head
1002 end

```

## 10.12 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.<sup>13</sup>

```

1003 hammertimedelay = 1.2
1004 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
1005 hammertime = function(head)
1006   if hammerfirst then

```

---

<sup>13</sup><http://tug.org/pipermail/luatex/2011-November/003355.html>

```

1007     texiowrite_nl(htime_separator)
1008     texiowrite_nl("=====STOP!=====\\n")
1009     texiowrite_nl(htime_separator .. "\\n\\n\\n")
1010     os.sleep (hammertimedelay*1.5)
1011     texiowrite_nl(htime_separator .. "\\n")
1012     texiowrite_nl("=====HAMMERTIME=====\\n")
1013     texiowrite_nl(htime_separator .. "\\n\\n")
1014     os.sleep (hammertimedelay)
1015     hammerfirst = false
1016 else
1017     os.sleep (hammertimedelay)
1018     texiowrite_nl(htime_separator)
1019     texiowrite_nl("=====U can't touch this!=====\\n")
1020     texiowrite_nl(htime_separator .. "\\n\\n")
1021     os.sleep (hammertimedelay*0.5)
1022 end
1023 return head
1024 end

```

### 10.13 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.

```

1025 itsame = function()
1026 local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
1027 color = "1 .6 0"
1028 for i = 6,9 do mr(i,3) end
1029 for i = 3,11 do mr(i,4) end
1030 for i = 3,12 do mr(i,5) end
1031 for i = 4,8 do mr(i,6) end
1032 for i = 4,10 do mr(i,7) end
1033 for i = 1,12 do mr(i,11) end
1034 for i = 1,12 do mr(i,12) end
1035 for i = 1,12 do mr(i,13) end
1036
1037 color = ".3 .5 .2"
1038 for i = 3,5 do mr(i,3) end mr(8,3)
1039 mr(2,4) mr(4,4) mr(8,4)
1040 mr(2,5) mr(4,5) mr(5,5) mr(9,5)
1041 mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
1042 for i = 3,8 do mr(i,8) end
1043 for i = 2,11 do mr(i,9) end
1044 for i = 1,12 do mr(i,10) end
1045 mr(3,11) mr(10,11)
1046 for i = 2,4 do mr(i,15) end for i = 9,11 do mr(i,15) end
1047 for i = 1,4 do mr(i,16) end for i = 9,12 do mr(i,16) end

```

```

1048
1049 color = "1 0 0"
1050 for i = 4,9 do mr(i,1) end
1051 for i = 3,12 do mr(i,2) end
1052 for i = 8,10 do mr(5,i) end
1053 for i = 5,8 do mr(i,10) end
1054 mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
1055 for i = 4,9 do mr(i,12) end
1056 for i = 3,10 do mr(i,13) end
1057 for i = 3,5 do mr(i,14) end
1058 for i = 7,10 do mr(i,14) end
1059 end

```

## 10.14 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.

```

1060 chickenkernamount = 0
1061 chickeninvertkerning = false
1062
1063 function kernmanipulate (head)
1064   if chickeninvertkerning then -- invert the kerning
1065     for n in nodetraverseid(11,head) do
1066       n.kern = -n.kern
1067     end
1068   else -- if not, set it to the given value
1069     for n in nodetraverseid(11,head) do
1070       n.kern = chickenkernamount
1071     end
1072   end
1073   return head
1074 end

```

## 10.15 leetspeak

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

```

1075 leetspeak_onlytext = false
1076 leettable = {
1077   [101] = 51, -- E
1078   [105] = 49, -- I
1079   [108] = 49, -- L
1080   [111] = 48, -- O

```

```

1081 [115] = 53, -- S
1082 [116] = 55, -- T
1083
1084 [101-32] = 51, -- e
1085 [105-32] = 49, -- i
1086 [108-32] = 49, -- l
1087 [111-32] = 48, -- o
1088 [115-32] = 53, -- s
1089 [116-32] = 55, -- t
1090 }

```

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

```

1091 leet = function(head)
1092   for line in nodetraverseid(Hhead,head) do
1093     for i in nodetraverseid(GLYPH,line.head) do
1094       if not leetspeak_onlytext or
1095         node.has_attribute(i,luatexbase.attributes.leetattr)
1096       then
1097         if leettable[i.char] then
1098           i.char = leettable[i.char]
1099         end
1100       end
1101     end
1102   end
1103   return head
1104 end

```

## 10.16 leftsideright

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

```

1105 leftsideright = function(head)
1106   local factor = 65536/0.99626
1107   for n in nodetraverseid(GLYPH,head) do
1108     if (leftsiderightarray[n.char]) then
1109       shift = nodenew(WHAT,PDF_LITERAL)
1110       shift2 = nodenew(WHAT,PDF_LITERAL)
1111       shift.data = "q -1 0 0 1 " .. n.width/factor .. " 0 cm"
1112       shift2.data = "Q 1 0 0 1 " .. n.width/factor .. " 0 cm"
1113       nodeinsertbefore(head,n,shift)
1114       nodeinsertafter(head,n,shift2)
1115     end
1116   end
1117   return head
1118 end

```

## 10.17 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>

### 10.17.1 setup of variables

```
1119 local letterspace_glue = nodenew(GLUE)
1120 local letterspace_pen   = nodenew(PENALTY)
1121
1122 letterspace_glue.width   = tex.sp"0pt"
1123 letterspace_glue.stretch = tex.sp"0.5pt"
1124 letterspace_pen.penalty  = 10000
```

### 10.17.2 function implementation

```
1125 letterspaceadjust = function(head)
1126   for glyph in nodetraverseid(GLYPH, head) do
1127     if glyph.prev and (glyph.prev.id == GLYPH or glyph.prev.id == DISC or glyph.prev.id == KERN) then
1128       local g = nodecopy(letterspace_glue)
1129       nodeinsertbefore(head, glyph, g)
1130       nodeinsertbefore(head, g, nodecopy(letterspace_pen))
1131     end
1132   end
1133   return head
1134 end
```

### 10.17.3 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!

```
1135 textletterspaceadjust = function(head)
1136   for glyph in nodetraverseid(GLYPH, head) do
1137     if node.has_attribute(glyph, luatexbase.attributes.letterspaceadjustattr) then
1138       if glyph.prev and (glyph.prev.id == node.id"glyph" or glyph.prev.id == node.id"disc" or glyph.prev.id == node.id"kern") then
1139         local g = node.copy(letterspace_glue)
1140         nodeinsertbefore(head, glyph, g)
1141         nodeinsertbefore(head, g, nodecopy(letterspace_pen))
1142       end
1143     end
1144   end
1145   luatexbase.remove_from_callback("pre_linebreak_filter", "textletterspaceadjust")
1146   return head
```

1147 end

### 10.18 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.

```
1148 matrixize = function(head)
1149   x = {}
1150   s = nodenew(DISC)
1151   for n in nodetraverseid(GLYPH,head) do
1152     j = n.char
1153     for m = 0,7 do -- stay ASCII for now
1154       x[7-m] = nodecopy(n) -- to get the same font etc.
1155     end
1156     if (j / (2^(7-m)) < 1) then
1157       x[7-m].char = 48
1158     else
1159       x[7-m].char = 49
1160       j = j-(2^(7-m))
1161     end
1162     nodeinsertbefore(head,n,x[7-m])
1163     nodeinsertafter(head,x[7-m],nodecopy(s))
1164   end
1165   noderemove(head,n)
1166 end
1167 return head
1168 end
```

### 10.19 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 ...

```
1169 medievalumlaut = function(head)
1170   local factor = 65536/0.99626
1171   local org_e_node = nodenew(GLYPH)
1172   org_e_node.char = 101
1173   for line in nodetraverseid(0,head) do
1174     for n in nodetraverseid(GLYPH,line.head) do
1175       if (n.char == 228 or n.char == 246 or n.char == 252) then
1176         e_node = nodecopy(org_e_node)
1177         e_node.font = n.font
```

```

1178     shift = nodenew(WHAT,PDF_LITERAL)
1179     shift2 = nodenew(WHAT,PDF_LITERAL)
1180     shift2.data = "Q 1 0 0 1 " .. e_node.width/factor .. " 0 cm"
1181     nodeinsertafter(head,n,e_node)
1182
1183     nodeinsertbefore(head,e_node,shift)
1184     nodeinsertafter(head,e_node,shift2)
1185
1186     x_node = nodenew(KERN)
1187     x_node.kern = -e_node.width
1188     nodeinsertafter(head,shift2,x_node)
1189 end
1190
1191 if (n.char == 228) then -- ä
1192     shift.data = "q 0.5 0 0 0.5 " ..
1193         -n.width/factor*0.85 .. " .. n.height/factor*0.75 .. " cm"
1194     n.char = 97
1195 end
1196 if (n.char == 246) then -- ö
1197     shift.data = "q 0.5 0 0 0.5 " ..
1198         -n.width/factor*0.75 .. " .. n.height/factor*0.75 .. " cm"
1199     n.char = 111
1200 end
1201 if (n.char == 252) then -- ü
1202     shift.data = "q 0.5 0 0 0.5 " ..
1203         -n.width/factor*0.75 .. " .. n.height/factor*0.75 .. " cm"
1204     n.char = 117
1205 end
1206 end
1207 end
1208 return head
1209 end

```

## 10.20 pancakenize

```

1210 local separator      = string.rep("=", 28)
1211 local texiowrite_nl = texio.write_nl
1212 pancaketext = function()
1213     texiowrite_nl("Output written on "..tex.jobname.." pdf ("..status.total_pages.." chicken,".." eg
1214     texiowrite_nl(" ")
1215     texiowrite_nl(separator)
1216     texiowrite_nl("Soo ... you decided to use \\pancakenize.")
1217     texiowrite_nl("That means you owe me a pancake!")
1218     texiowrite_nl(" ")
1219     texiowrite_nl("(This goes by document, not compilation.)")
1220     texiowrite_nl(separator.."\\n\\n")

```

```

1221 texiowrite_nl("Looking forward for my pancake! :)")
1222 texiowrite_nl("\n\n")
1223 end

```

### 10.21 randomerror

Not yet implemented, sorry.

### 10.22 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.

```

1224 randomfontslower = 1
1225 randomfontsupper = 0
1226 %
1227 randomfonts = function(head)
1228   local rfub
1229   if randomfontsupper > 0 then -- fixme: this should be done only once, no? Or at every paragraph
1230     rfub = randomfontsupper -- user-specified value
1231   else
1232     rfub = font.max() -- or just take all fonts
1233   end
1234   for line in nodetraverseid(Hhead,head) do
1235     for i in nodetraverseid(GLYPH,line.head) do
1236       if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattrib) then
1237         i.font = math.random(randomfontslower,rfub)
1238       end
1239     end
1240   end
1241   return head
1242 end

```

### 10.23 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```

1243 uclcratio = 0.5 -- ratio between uppercase and lower case
1244 randomuclc = function(head)
1245   for i in nodetraverseid(GLYPH,head) do
1246     if not(randomuclc_onlytext) or node.has_attribute(i,luatexbase.attributes.randuclcattrib) then
1247       if math.random() < uclcratio then
1248         i.char = tex.uccode[i.char]
1249       else
1250         i.char = tex.lccode[i.char]
1251       end
1252     end
1253   end
1254   return head

```



```
1255 end
```

## 10.24 randomchars

```
1256 randomchars = function(head)
1257   for line in nodetraverseid(Hhead,head) do
1258     for i in nodetraverseid(GLYPH,line.head) do
1259       i.char = math.floor(math.random()*512)
1260     end
1261   end
1262   return head
1263 end
```

## 10.25 randomcolor and rainbowcolor

### 10.25.1 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.

```
1264 randomcolor_grey = false
1265 randomcolor_onlytext = false --switch between local and global colorization
1266 rainbowcolor = false
1267
1268 grey_lower = 0
1269 grey_upper = 900
1270
1271 Rgb_lower = 1
1272 rGb_lower = 1
1273 rgB_lower = 1
1274 Rgb_upper = 254
1275 rGb_upper = 254
1276 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.

```
1277 rainbow_step = 0.005
1278 rainbow_Rgb = 1-rainbow_step -- we start in the red phase
1279 rainbow_rGb = rainbow_step -- values x must always be  $0 < x < 1$ 
1280 rainbow_rgB = rainbow_step
1281 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.

```
1282 randomcolorstring = function()
1283   if randomcolor_grey then
1284     return (0.001*math.random(grey_lower,grey_upper)).." g"
1285   elseif rainbowcolor then
1286     if rainind == 1 then -- red
1287       rainbow_rGb = rainbow_rGb + rainbow_step
```

```

1288     if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
1289 elseif rainind == 2 then -- yellow
1290     rainbow_Rgb = rainbow_Rgb - rainbow_step
1291     if rainbow_Rgb <= rainbow_step then rainind = 3 end
1292 elseif rainind == 3 then -- green
1293     rainbow_rGb = rainbow_rGb + rainbow_step
1294     rainbow_rGb = rainbow_rGb - rainbow_step
1295     if rainbow_rGb <= rainbow_step then rainind = 4 end
1296 elseif rainind == 4 then -- blue
1297     rainbow_Rgb = rainbow_Rgb + rainbow_step
1298     if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
1299 else -- purple
1300     rainbow_rGb = rainbow_rGb - rainbow_step
1301     if rainbow_rGb <= rainbow_step then rainind = 1 end
1302 end
1303 return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rGb.." rg"
1304 else
1305     Rgb = math.random(Rgb_lower,Rgb_upper)/255
1306     rGb = math.random(rGb_lower,rGb_upper)/255
1307     rgB = math.random(rgB_lower,rgB_upper)/255
1308     return Rgb.." "..rGb.." "..rgB.." ".." rg"
1309 end
1310 end

```

### 10.25.2 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.

```

1311 randomcolor = function(head)
1312   for line in nodetraverseid(0,head) do
1313     for i in nodetraverseid(GLYPH,line.head) do
1314       if not(randomcolor_onlytext) or
1315         (node.has_attribute(i,luatexbase.attributes.randcolorattr))
1316       then
1317         color_push.data = randomcolorstring() -- color or grey string
1318         line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
1319         nodeinsertafter(line.head,i,nodecopy(color_pop))
1320       end
1321     end
1322   end
1323   return head
1324 end

```

### 10.26 randomerror

1325 %

## 10.27 relationship

It literally is what it says: A ship made of relations. Or a boat, rather. There are four parameters, `sailheight`, `mastheight`, `hullheight`, and `relnumber` which you can adjust.

```
1326 function relationship()
1327   sailheight = 12
1328   mastheight = 4
1329   hullheight = 5
1330   relnumber = 402
1331   shipheight = sailheight + mastheight + hullheight
1332   tex.print("\parshape " .. (shipheight))
1333   for i = 1, sailheight do
1334     tex.print(" " .. (4.5 - i / 3.8) .. "cm " .. ((i - 0.5) / 2.5) .. "cm ")
1335   end
1336   for i = 1, mastheight do
1337     tex.print(" " .. (3.2) .. "cm " .. (1) .. "cm ")
1338   end
1339   for i = 1, hullheight do
1340     tex.print(" " .. ((i - 1) / 2) .. "cm " .. (10 - i) .. "cm ")
1341   end
1342   tex.print("\noindent")
1343   for i = 1, relnumber do
1344     tex.print("\ \char" .. math.random(8756, 8842))
1345   end
1346   tex.print("\break")
1347 end
```

And this is a helper function to prevent too many relations to be typeset. Problem: The relations are chosen randomly, and each might take different horizontal space. So we cannot make sure the same number of lines for each version. To catch this, we typeset more lines and just remove excess lines with a simple function in our beloved `post_linebreak_filter`.

```
1348 function cutparagraph(head)
1349   local parsum = 0
1350   for n in nodetraverseid(HLIST, head) do
1351     parsum = parsum + 1
1352     if parsum > shipheight then
1353       node.remove(head, n)
1354     end
1355   end
1356   return head
1357 end
```

## 10.28 rickroll

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

1358 %

## 10.29 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.

```
1359 substitutewords_strings = {}
1360
1361 addtosubstitutions = function(input,output)
1362   substitutewords_strings[#substitutewords_strings + 1] = {}
1363   substitutewords_strings[#substitutewords_strings][1] = input
1364   substitutewords_strings[#substitutewords_strings][2] = output
1365 end
1366
1367 substitutewords = function(head)
1368   for i = 1,#substitutewords_strings do
1369     head = string.gsub(head,substitutewords_strings[i][1],substitutewords_strings[i][2])
1370   end
1371   return head
1372 end
```

## 10.30 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 wether 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.

```
1373 suppressonecharbreakpenaltynode = node.new(PENALTY)
1374 suppressonecharbreakpenaltynode.penalty = 10000
1375 function suppressonecharbreak(head)
1376   for i in node.traverse_id(GLUE,head) do
1377     if ((i.next) and (i.next.next.id == GLUE)) then
1378       pen = node.copy(suppressonecharbreakpenaltynode)
1379       node.insert_after(head,i.next,pen)
1380     end
```

```

1381 end
1382
1383 return head
1384 end

```

### 10.31 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.

```

1385 tabularasa_onlytext = false
1386
1387 tabularasa = function(head)
1388   local s = nodenew(KERN)
1389   for line in nodetraverseid(HLIST,head) do
1390     for n in nodetraverseid(GLYPH,line.head) do
1391       if not(tabularasa_onlytext) or node.has_attribute(n,luatexbase.attributes.tabularasaattr) then
1392         s.kern = n.width
1393         nodeinsertafter(line.list,n,nodecopy(s))
1394         line.head = noderemove(line.list,n)
1395       end
1396     end
1397   end
1398   return head
1399 end

```

### 10.32 tanjanize

```

1400 tanjanize = function(head)
1401   local s = nodenew(KERN)
1402   local m = nodenew(GLYPH,1)
1403   local use_letter_i = true
1404   scale = nodenew(WHAT,PDF_LITERAL)
1405   scale2 = nodenew(WHAT,PDF_LITERAL)
1406   scale.data = "0.5 0 0 0.5 0 0 cm"
1407   scale2.data = "2 0 0 2 0 0 cm"
1408
1409   for line in nodetraverseid(HLIST,head) do
1410     for n in nodetraverseid(GLYPH,line.head) do
1411       mimicount = 0
1412       tmpwidth = 0
1413       while ((n.next.id == GLYPH) or (n.next.id == 11) or (n.next.id == 7) or (n.next.id == 0)) do
1414         find end of a word
1415         n.next = n.next.next
1416         mimicount = mimicount + 1
1417         tmpwidth = tmpwidth + n.width
1418       end

```

```

1418
1419     mimi = {} -- constructing the node list.
1420     mimi[0] = nodenew(GLYPH,1) -- only a dummy for the loop
1421     for i = 1,string.len(mimicount) do
1422         mimi[i] = nodenew(GLYPH,1)
1423         mimi[i].font = font.current()
1424         if(use_letter_i) then mimi[i].char = 109 else mimi[i].char = 105 end
1425         use_letter_i = not(use_letter_i)
1426         mimi[i-1].next = mimi[i]
1427     end
1428 --]]
1429
1430 line.head = nodeinsertbefore(line.head,n,nodecopy(scale))
1431 nodeinsertafter(line.head,n,nodecopy(scale2))
1432     s.kern = (tmpwidth*2-n.width)
1433     nodeinsertafter(line.head,n,nodecopy(s))
1434 end
1435 end
1436 return head
1437 end

```

### 10.33 uppercasecolor

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

```

1438 uppercasecolor_onlytext = false
1439
1440 uppercasecolor = function (head)
1441     for line in nodetraverseid(Hhead,head) do
1442         for upper in nodetraverseid(GLYPH,line.head) do
1443             if not(uppercasecolor_onlytext) or node.has_attribute(upper,luatexbase.attributes.uppercasecolor) then
1444                 if (((upper.char > 64) and (upper.char < 91)) or
1445                     ((upper.char > 57424) and (upper.char < 57451))) then -- for small caps! nice
1446                     color_push.data = randomcolorstring() -- color or grey string
1447                     line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
1448                     nodeinsertafter(line.head,upper,nodecopy(color_pop))
1449                 end
1450             end
1451         end
1452     end
1453     return head
1454 end

```

### 10.34 upsidedown

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

```

1455 upsidedown = function(head)
1456     local factor = 65536/0.99626

```

```

1457 for line in nodetraverseid(Hhead,head) do
1458   for n in nodetraverseid(GLYPH,line.head) do
1459     if (upsidedownarray[n.char]) then
1460       shift = nodenew(WHAT,PDF_LITERAL)
1461       shift2 = nodenew(WHAT,PDF_LITERAL)
1462       shift.data = "q 1 0 0 -1 0 " .. n.height/factor .." cm"
1463       shift2.data = "Q 1 0 0 1 " .. n.width/factor .." 0 cm"
1464       nodeinsertbefore(head,n,shift)
1465       nodeinsertafter(head,n,shift2)
1466     end
1467   end
1468 end
1469 return head
1470 end

```

## 10.35 colorstretch

This function displays the amount of stretching that has been done for each line of an arbitrary document. A well-typeset document should be equally grey over all lines, which is not always possible.

In fact, two boxes are drawn: The first (left) box shows the badness, i. e. the amount of stretching the spaces between words. Too much space results in light grey, whereas a too dense line is indicated by a dark grey box.

The second box is only useful if microtypographic extensions are used, e. g. with the microtype package under  $\TeX$ . The box color then corresponds to the amount of font expansion in the line. This works great for demonstrating the positive effect of font expansion on the badness of a line!

The base structure of the following code was provided by Paul Isambert. Thanks for the code and support, Paul!

### 10.35.1 colorstretch – preliminaries

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

```

1471 keeptext = true
1472 colorexpansion = true
1473
1474 colorstretch_coloroffset = 0.5
1475 colorstretch_colorrage = 0.5
1476 chickenize_rule_bad_height = 4/5 -- height and depth of the rules
1477 chickenize_rule_bad_depth = 1/5
1478
1479
1480 colorstretchnumbers = true
1481 drawstretchthreshold = 0.1
1482 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 (`colorexansion == 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.

```

1483 colorstretch = function (head)
1484   local f = font.getfont(font.current()).characters
1485   for line in nodetraverseid(Hhead,head) do
1486     local rule_bad = nodenew(RULE)
1487
1488     if colorexpansion then -- if also the font expansion should be shown
1489 --%% here use first_glyph function!!
1490       local g = line.head
1491       n = node.first_glyph(line.head.next)
1492       texio.write_nl(line.head.id)
1493       texio.write_nl(line.head.next.id)
1494       texio.write_nl(line.head.next.next.id)
1495       texio.write_nl(n.id)
1496       while not(g.id == GLYPH) and (g.next) do g = g.next end -- find first glyph on line. If line
1497       if (g.id == GLYPH) then -- read width only if g is a glyph!
1498         exp_factor = g.expansion_factor/10000 --%% neato, luatex now directly gives me this!!
1499         exp_color = colorstretch_coloroffset + (exp_factor*0.1) .. " g"
1500       texio.write_nl(exp_factor)
1501       rule_bad.width = 0.5*line.width -- we need two rules on each line!
1502     end
1503   else
1504     rule_bad.width = line.width -- only the space expansion should be shown, only one rule
1505   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.

```

1506   rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bet
1507   rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth
1508
1509   local glue_ratio = 0
1510   if line.glue_order == 0 then
1511     if line.glue_sign == 1 then
1512       glue_ratio = colorstretch_colorrange * math.min(line.glue_set,1)
1513     else
1514       glue_ratio = -colorstretch_colorrange * math.min(line.glue_set,1)
1515     end
1516   end
1517   color_push.data = colorstretch_coloroffset + glue_ratio .. " g"
1518

```

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

```

1519 -- set up output
1520   local p = line.head
1521

```



```

1522 -- a rule to immitate kerning all the way back
1523     local kern_back = nodenew(RULE)
1524     kern_back.width = -line.width
1525
1526 -- if the text should still be displayed, the color and box nodes are inserted additionally
1527 -- and the head is set to the color node
1528     if keeptext then
1529         line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1530     else
1531         node.flush_list(p)
1532         line.head = nodecopy(color_push)
1533     end
1534     nodeinsertafter(line.head,line.head,rule_bad) -- then the rule
1535     nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
1536     tmpnode = nodeinsertafter(line.head,line.head.next.next,kern_back)
1537
1538 -- then a rule with the expansion color
1539 if colorexansion then -- if also the stretch/shrink of letters should be shown
1540     color_push.data = exp_color
1541     nodeinsertafter(line.head,tmpnode,nodecopy(color_push))
1542     nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
1543     nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
1544 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.

```

1545 if colorstretchnumbers then
1546     j = 1
1547     glue_ratio_output = {}
1548     for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
1549         local char = unicode.utf8.char(s)
1550         glue_ratio_output[j] = nodenew(GLYPH,1)
1551         glue_ratio_output[j].font = font.current()
1552         glue_ratio_output[j].char = s
1553         j = j+1
1554     end
1555     if math.abs(glue_ratio) > drawstretchthreshold then
1556         if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
1557         else color_push.data = "0 0.99 0 rg" end
1558     else color_push.data = "0 0 0 rg"
1559     end
1560
1561     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
1562     for i = 1,math.min(j-1,7) do
1563         nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])

```

```

1564     end
1565     nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
1566 end -- end of stretch number insertion
1567 end
1568 return head
1569 end

```

## dubstepize

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

```

1570

```

## scorpionize

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

```

1571 function scorpionize_color(head)
1572   color_push.data = ".35 .55 .75 rg"
1573   nodeinsertafter(head,head,nodecopy(color_push))
1574   nodeinsertafter(head,node.tail(head),nodecopy(color_pop))
1575   return head
1576 end

```

## 10.36 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 ... (?)

```

1577 substlist = {}
1578 substlist[1488] = 64289
1579 substlist[1491] = 64290
1580 substlist[1492] = 64291
1581 substlist[1499] = 64292
1582 substlist[1500] = 64293
1583 substlist[1501] = 64294
1584 substlist[1512] = 64295
1585 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").

```

1586 function variantjustification(head)

```

```

1587 math.randomseed(1)
1588 for line in nodetraverseid(Hhead,head) do
1589     if (line.glue_sign == 1 and line.glue_order == 0) then -- exclude the last line!
1590         substitutions_wide = {} -- we store all "expandable" letters of each line
1591         for n in nodetraverseid(GLYPH,line.head) do
1592             if (substlist[n.char]) then
1593                 substitutions_wide[#substitutions_wide+1] = n
1594             end
1595         end
1596         line.glue_set = 0 -- deactivate normal glue expansion
1597         local width = node.dimensions(line.head) -- check the new width of the line
1598         local goal = line.width
1599         while (width < goal and #substitutions_wide > 0) do
1600             x = math.random(#substitutions_wide) -- choose randomly a glyph to be substituted
1601             oldchar = substitutions_wide[x].char
1602             substitutions_wide[x].char = substlist[substitutions_wide[x].char] -- substitute by wide
1603             width = node.dimensions(line.head) -- check if the line is too wide
1604             if width > goal then substitutions_wide[x].char = oldchar break end -- substitute back if
1605             table.remove(substitutions_wide,x) -- if further substitutions have to be done,
1606         end
1607     end
1608 end
1609 return head
1610 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.

## 10.37 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`.

### 10.37.1 zebranize – preliminaries

```

1611 zebracolorarray = {}
1612 zebracolorarray_bg = {}
1613 zebracolorarray[1] = "0.1 g"
1614 zebracolorarray[2] = "0.9 g"
1615 zebracolorarray_bg[1] = "0.9 g"
1616 zebracolorarray_bg[2] = "0.1 g"

```

### 10.37.2 zebranize – the function

This code has to be revisited, it is ugly.

```
1617 function zebranize(head)
1618   zebracolor = 1
1619   for line in nodetraverseid(Hhead,head) do
1620     if zebracolor == #zebracolorarray then zebracolor = 0 end
1621     zebracolor = zebracolor + 1
1622     color_push.data = zebracolorarray[zebracolor]
1623     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1624     for n in nodetraverseid(GLYPH,line.head) do
1625       if n.next then else
1626         nodeinsertafter(line.head,n,nodecopy(color_pull))
1627       end
1628     end
1629
1630     local rule_zebra = nodenew(RULE)
1631     rule_zebra.width = line.width
1632     rule_zebra.height = tex.baselineskip.width*4/5
1633     rule_zebra.depth = tex.baselineskip.width*1/5
1634
1635     local kern_back = nodenew(RULE)
1636     kern_back.width = -line.width
1637
1638     color_push.data = zebracolorarray_bg[zebracolor]
1639     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
1640     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1641     nodeinsertafter(line.head,line.head,kern_back)
1642     nodeinsertafter(line.head,line.head,rule_zebra)
1643   end
1644   return (head)
1645 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! The parameters `sloppinessh` and `sloppinessv` give the amount of sloppiness, i. e. how strongly the points are “wiggled” randomly to make the drawings more dynamically. You can set them at any time in the document

```

1646 --
1647 function pdf_print (...)
1648   for _, str in ipairs({...}) do
1649     pdf.print(str .. " ")
1650   end
1651   pdf.print("\n")
1652 end
1653
1654 function move (p)
1655   pdf_print(p[1],p[2],"m")
1656 end
1657
1658 function line (p)
1659   pdf_print(p[1],p[2],"l")
1660 end
1661
1662 function curve(p1,p2,p3)
1663   pdf_print(p1[1], p1[2],
1664             p2[1], p2[2],
1665             p3[1], p3[2], "c")
1666 end
1667
1668 function close ()
1669   pdf_print("h")
1670 end
1671
1672 function linewidth (w)
1673   pdf_print(w,"w")
1674 end
1675
1676 function stroke ()
1677   pdf_print("S")

```

```

1678 end
1679 --
1680
1681 function strictcircle(center,radius)
1682   local left = {center[1] - radius, center[2]}
1683   local lefttop = {left[1], left[2] + 1.45*radius}
1684   local leftbot = {left[1], left[2] - 1.45*radius}
1685   local right = {center[1] + radius, center[2]}
1686   local righttop = {right[1], right[2] + 1.45*radius}
1687   local rightbot = {right[1], right[2] - 1.45*radius}
1688
1689   move (left)
1690   curve (lefttop, righttop, right)
1691   curve (rightbot, leftbot, left)
1692 stroke()
1693 end
1694
1695 sloppynessh = 5
1696 sloppynessv = 5
1697
1698 function disturb_point(point)
1699   return {point[1] + (math.random() - 1/2)*sloppynessh,
1700           point[2] + (math.random() - 1/2)*sloppynessv}
1701 end
1702
1703 function sloppycircle(center,radius)
1704   local left = disturb_point({center[1] - radius, center[2]})
1705   local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
1706   local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}
1707   local right = disturb_point({center[1] + radius, center[2]})
1708   local righttop = disturb_point({right[1], right[2] + 1.45*radius})
1709   local rightbot = disturb_point({right[1], right[2] - 1.45*radius})
1710
1711   local right_end = disturb_point(right)
1712
1713   move (right)
1714   curve (rightbot, leftbot, left)
1715   curve (lefttop, righttop, right_end)
1716   linewidth(math.random()+0.5)
1717   stroke()
1718 end
1719
1720 function sloppyellipsis(center,radiusx,radiusy)
1721   local left = disturb_point({center[1] - radiusx, center[2]})
1722   local lefttop = disturb_point({left[1], left[2] + 1.45*radiusy})
1723   local leftbot = {lefttop[1], lefttop[2] - 2.9*radiusy}

```

```

1724 local right = disturb_point({center[1] + radiusx, center[2]})
1725 local righttop = disturb_point({right[1], right[2] + 1.45*radiusy})
1726 local rightbot = disturb_point({right[1], right[2] - 1.45*radiusy})
1727
1728 local right_end = disturb_point(right)
1729
1730 move (right)
1731 curve (rightbot, leftbot, left)
1732 curve (lefttop, righttop, right_end)
1733 linewidth(math.random()+0.5)
1734 stroke()
1735 end
1736
1737 function sloppyline(start,stop)
1738 local start_line = disturb_point(start)
1739 local stop_line = disturb_point(stop)
1740 start = disturb_point(start)
1741 stop = disturb_point(stop)
1742 move(start) curve(start_line,stop_line,stop)
1743 linewidth(math.random()+0.5)
1744 stroke()
1745 end

```

## 12 Known Bugs and Fun Facts

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.

**medievalumlaut** You should use a decent OpenType font to get the best result. The standard font will not nicely support the positioning of the `e` character.

**boustrophedon and chickenize** do not work together nicely. There is an additional shift I cannot explain so far. However, if you really, really need a boustrophedon of `chickenize`, you do have some serious problems.

**letterspaceadjust and chickenize** When using both `letterspaceadjust` and `chickenize`, make sure to activate `\chickenize` before `\letterspaceadjust`. Elsewise the chickenization will not work due to the implementation of `letterspaceadjust`.

## 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 differentiate 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<sub>T</sub><sub>E</sub>X 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, 1<sup>st</sup> 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<sub>T</sub><sub>E</sub>X 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 ...