



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

Paul Isambert

CHICKENIZE

v0.3a

Arno L. Trautmann \AA

arno.trautmann@gmx.de

How to read this document.

This is the documentation of the package `chickenize`. It allows manipulations of any Lua \TeX document¹ exploiting the possibilities offered by the callbacks that influence line breaking (and some other stuff). Most of this package's content is just for fun and educational use, but there are also some functions that can be useful in a normal 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 \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 © 2021 Arno L. Trautmann. It may be distributed and/or modified under the conditions of the LaTeX Project Public License, either version 1.3c of this license or (at your option) any later version. This work has the LPPL maintenance status 'maintained'.

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

For the Impatient:

A small and incomplete overview of the functionalities offered by this package.² 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

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

less useful functions

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

complete nonsense

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

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

Contents

I	User Documentation	6
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 chickenization	10
3.2	Options for Game of Chicken	11
II	Tutorial	13
4	Lua code	13
5	callbacks	13
6	How to use a callback	14
7	Nodes	14
8	Other things	15
III	Implementation	16
9	TeX file	16
9.1	allownumberincommands	16
9.2	drawchicken	26
9.3	drawcov	27
9.4	drawhorse	27
10	LaTeX package	29
10.1	Free Compliments	29
10.2	Definition of User-Level Macros	29
11	Lua Module	29
11.1	chickenize	30
11.2	shownodes	33
11.3	boustrophedon	34
11.4	bubblesort	35

11.5	countglyphs	36
11.6	countwords	36
11.7	detectdoublewords	37
11.8	francize	37
11.9	gameofchicken	38
11.10	gutenbergize	40
11.10.1	gutenbergize – preliminaries	40
11.10.2	gutenbergize – the function	40
11.11	hammertime	41
11.12	italianize	41
11.13	italianizerandwords	43
11.14	itsame	44
11.15	kernmanipulate	45
11.16	leetspeak	45
11.17	leftsideright	46
11.18	letterspaceadjust	47
11.18.1	setup of variables	47
11.18.2	function implementation	47
11.18.3	textletterspaceadjust	47
11.19	matrixize	48
11.20	medievalumlaut	48
11.21	pancakenize	49
11.22	randomerror	50
11.23	randomfonts	50
11.24	randomucl	50
11.25	randomchars	51
11.26	randomcolor and rainbowcolor	51
11.26.1	randomcolor – preliminaries	51
11.26.2	randomcolor – the function	52
11.27	relationship	53
11.28	rickroll	54
11.29	substitutewords	54
11.30	suppressonecharbreak	55
11.31	tabularasa	55
11.32	tanjanize	56
11.33	uppercasecolor	56
11.34	upsidedown	57
11.35	colorstretch	57
11.35.1	colorstretch – preliminaries	58
11.36	variantjustification	60
11.37	zebranize	61
11.37.1	zebranize – preliminaries	62
11.37.2	zebranize – the function	62
12	Drawing	64

13 Known Bugs and Fun Facts	67
14 To Do's	67
15 Literature	67
16 Thanks	68

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 provided 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. If no link is provided then the command is mostly just an adaption of another one. I'll try to get this consistent somehow, but for now, it's not.

`\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 imitates archaic greek writings where one line was right-to-left, the next one left-to-right etc.³ Interestingly, also every glyph was adapted 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 Rongorongong similar style boustrophedon is available with `\boustrophedoninverse` or `\rongorongonize`, where subsequent lines are rotated by 180° instead of mirrored.

³en.wikipedia.org/wiki/Boustrophedon

⁴en.wikipedia.org/wiki/Rongorongong

\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 10th chicken is uppercase. However, the beginning of a sentence is not recognized automatically.⁵

\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. It’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 L^AT_EX 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, learn 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.

⁵If 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 its functionality. So once you used `\chickenize`, it will chickenize the whole document up to `\unchickenize`. However, the paragraph in which `\unchickenize` appears, will *not* be chickenized. The same is true for all other manipulations. Take care that you don't `\un-`anything before activating it, as this will result in an error.⁶

If you want to manipulate only a part of a paragraph, you will have to use the corresponding `\text-`version of the function, see below. However, feel free to set and unset every function at will at any place in your document.

2.3 `\text-Versions`

The functions provided by this package might be much more useful if applied only to a short sequence of words or single words instead of the whole document or paragraph. Therefore, most of the above-mentioned commands have⁷ a `\text-`version that takes an argument. `\textrandomcolor{foo}` results in a colored `foo` while the rest of the document remains unaffected. However, to achieve this effect, still the whole node list has to be traversed. Thus, it may slow down the compilation of your document, even if you use `\textrandomcolor` only once. Fortunately, the effect is very small and mostly negligible.⁸

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

2.4 Lua functions

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

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

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

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

3 Options – How to Adjust It

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

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

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

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

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

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

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

3.1 options for chickenization

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

`colorstretchnumbers` = **<bool>** `false` 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`).

`drawwidth` = **<float>** 1 Defines the widths of the sloppy drawings of chickens, horses, etc.

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

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

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

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

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

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

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

3.2 Options for Game of Chicken

test This deserves a separate section since there are some more options and they need some explanation. So here go 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 \LaTeX code for graphical representation of a living cell. You can use basically any valid \LaTeX code in here. A chicken is the default, of course.

GOCanticellcode = <string> "0" The \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 \TeX . It's just to get an idea how things work here. For a deeper understanding of Lua \TeX you should consult both the Lua \TeX manual and some introduction into Lua proper like "Programming in Lua". (See the section [Literature](#) at the end of the manual.)

4 Lua code

The crucial novelty in Lua \TeX is the first part of its name: The programming language Lua. One can use nearly any Lua code inside the commands `\directlua{}` or `\latelua{}`. This alleviates simple tasks like calculating a number and printing it, just as if it was entered by hand:

```
\directlua{
  a = 5*2
  tex.print(a)
}
```

A number of additions to the Lua language renders it particularly suitable for \TeX ing, especially the `tex.` library that offers access to \TeX internals. In the simple example above, the function `tex.print()` inserts its argument into the \TeX input stream, so the result of the calculation (10) is printed in the document.

Larger parts of Lua code should not be embedded in your \TeX code, but rather in a separate file. It can then be loaded using

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

If you use Lua \TeX , you can also use the `luacode` environment from the eponymous package.

5 callbacks

While Lua code can be inserted using `\directlua` at any point in the input, a very powerful concept allows to change the way \TeX behaves: The *callbacks*. A callback is a point where you can hook into \TeX 's working and do anything to it that may make sense – or not. (Thus maybe breaking your document completely ...)

Callbacks are employed at several stages of \TeX 's work – e. g. for font loading, paragraph breaking, shipping out etc. In this package, we make heavy use of mostly two callbacks: The `pre_linebreak_filter` and the `post_linebreak` filter. These callbacks are called just before (or after, resp.) \TeX breaks a paragraph into lines. Normally, these callbacks are empty, so they are a great playground. In between these callbacks, the `linebreak_filter` takes care of \TeX 's line breaking mechanism. We won't touch this as I have no idea of what's going on there ;)

6 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.⁹ 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!

7 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

⁹Since 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.

nodes in a list – e.g. all glyphs in a vertical list that also contains glue, rules etc. This is achieved by calling the function `node.traverse_id(GLYPH,head)`, with the first argument giving the respective id of the nodes.¹⁰

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

```
function remove_e(head)
  for n in node.traverse_id(GLYPH,head) do
    if n.char == 101 then
      node.remove(head,n)
    end
  end
  return head
end

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

Now, don’t read on, but try out this code by yourself! Change the number of the character to be removed, try to play around a bit. Also, try to remove the spaces between words. Those are glue nodes – look up their id in the Lua_T_EX manual! Then, you have to remove the `if n.char` condition on the third line of the listing, because glue nodes lack a `.char` field. If everything works, you should have an input consisting of only one long word. Congratulations!

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

8 Other things

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

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

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

¹⁰GLYPH 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 Lua_T_EX version. We will use this substitute throughout this document.

Part III

Implementation

9 T_EX file

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

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

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

```
1 \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 }
```

9.1 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

```

Game of life – did you expect something else? lol.

```

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\italianizerandwords{
226 \directlua{luatexbase.add_to_callback("pre_linebreak_filter",italianizerandwords,"italianizerandwords")}
227 \def\unitalianizerandwords{
228 \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","italianizerandwords")}}
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     luatexbase.add_to_callback("stop_run",missingcharstext,"charsmissing")
332     relationship()
333   }
334 }
335
336 \let\rongorongonize\boustrophedoninverse
337 \let\unrongorongonize\unboustrophedoninverse
338
339 \def\scorpionize{
340   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",scorpionize_color,"scorpionize_color")}}
341 \def\unscorpionize{
342   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","scorpionize_color")}}
343
344 \def\shownodes#1{
345   \directlua{

```

```

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

```



```

392 }
393 \def\unupsidedown{
394   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","upsidedown")}}
395
396 \def\variantjustification{
397   \directlua{luatexbase.add_to_callback("post_linebreak_filter",variantjustification,"variantjust.
398 \def\unvariantjustification{
399   \directlua{luatexbase.remove_from_callback("post_linebreak_filter","variantjustification")}}
400
401 \def\zebranize{
402   \directlua{luatexbase.add_to_callback("post_linebreak_filter",zebranize,"zebranize")}}
403 \def\unzebranize{
404   \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.

```

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

```

```

436 {\setluatexattribute\tabularasaattr{42}#1\unsetluatexattribute\tabularasaattr}
437 \long\def\textuppercasecolor#1%
438 {\setluatexattribute\uppercasecolorattr{42}#1\unsetluatexattribute\uppercasecolorattr}

```

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

```

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

```

9.2 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.

```

440 \long\def\luadraw#1#2{%
441   \vbox to #1bp{%
442     \vfil
443     \latelua{pdf_print("q") #2 pdf_print("Q")}%
444   }%
445 }
446 \long\def\drawchicken{
447   \luadraw{90}{
448     chickenhead      = {200,50} % chicken head center
449     chickenhead_rad = 20
450
451     neckstart = {215,35} % neck
452     neckstop  = {230,10} %
453
454     chickenbody      = {260,-10}
455     chickenbody_rad = 40
456     chickenleg = {
457       {{260,-50},{250,-70},{235,-70}},
458       {{270,-50},{260,-75},{245,-75}}
459     }
460
461     beak_top = {185,55}
462     beak_front = {165,45}
463     beak_bottom = {185,35}
464
465     wing_front = {260,-10}
466     wing_bottom = {280,-40}
467     wing_back = {275,-15}
468
469     sloppyline(chickenhead,chickenhead_rad) sloppyline(neckstart,neckstop)
470     sloppyline(chickenbody,chickenbody_rad)
471     sloppyline(chickenleg[1][1],chickenleg[1][2]) sloppyline(chickenleg[1][2],chickenleg[1][3])
472     sloppyline(chickenleg[2][1],chickenleg[2][2]) sloppyline(chickenleg[2][2],chickenleg[2][3])
473     sloppyline(beak_front,beak_top) sloppyline(beak_front,beak_bottom)
474     sloppyline(wing_front,wing_bottom) sloppyline(wing_back,wing_bottom)
475   }

```

```
476 }
```

9.3 drawcov

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

```
477 \long\def\drawcov{
478   \luadraw{90}{
479     covbody = {200,50}
480     covbody_rad = 50
481
482     covcrown_rad = 5
483     crownno = 13
484     for i=1,crownno do
485       crownpos = {covbody[1]+1.4*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+1.4*covbody_rad}
486       crownconnect = {covbody[1]+covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+covbody_rad}
487       sloppycircle(crownpos,covcrown_rad)
488       sloppyline(crownpos,crownconnect)
489     end
490
491     covcrown_rad = 6
492     crownno = 8
493     for i=1,crownno do
494       crownpos = {covbody[1]+0.8*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+0.8*covbody_rad}
495       crownconnect = {covbody[1]+0.5*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+0.5*covbody_rad}
496       sloppycircle(crownpos,covcrown_rad)
497       sloppyline(crownpos,crownconnect)
498     end
499
500     covcrown_rad = 8
501     sloppycircle(covbody,covcrown_rad)
502     sloppycircle(covbody,covbody_rad)
503     sloppyline(covbody,covbody)
504   }
505 }
```

9.4 drawhorse

Well ... guess what this does.

```
506 \long\def\drawhorse{
507   \luadraw{90}{
508     horsebod = {100,-40}
509     sloppyellipsis(horsebod,50,20)
510     horsehead = {20,0}
511     sloppyellipsis(horsehead,25,15)
512     sloppyline({35,-10},{50,-40})

```

```

513     sloppyline({45,5},{80,-25})
514     sloppyline({60,-50},{60,-90})
515     sloppyline({70,-50},{70,-90})
516     sloppyline({130,-50},{130,-90})
517     sloppyline({140,-50},{140,-90})
518     sloppyline({150,-40},{160,-60})
519     sloppyline({150,-38},{160,-58})
520     sloppyline({150,-42},{160,-62})
521     sloppyline({-5,-10},{10,-5})
522     sloppyellipsis({30,5},5,2)  %% it's an eye, aye?
523     sloppyline({27,15},{34,25})
524     sloppyline({34,25},{37,13})
525 }
526 }

```

There's also a version with a bit more ... meat to the bones:

```

527 \long\def\drawfathorse{
528   \luadraw{90}{
529     horsebod = {100,-40}
530     sloppyellipsis(horsebod,50,40)
531     horsehead = {20,0}
532     sloppyellipsis(horsehead,25,15)
533     sloppyline({35,-10},{50,-40})
534     sloppyline({45,5},{70,-15})
535     sloppyline({60,-70},{60,-90})
536     sloppyline({70,-70},{70,-90})
537     sloppyline({130,-70},{130,-90})
538     sloppyline({140,-70},{140,-90})
539     sloppyline({150,-40},{160,-60})
540     sloppyline({150,-38},{160,-58})
541     sloppyline({150,-42},{160,-62})
542     sloppyline({-5,-10},{10,-5})
543     sloppyellipsis({30,5},5,2)  %% it's an eye, aye?
544     sloppyline({27,15},{34,25})
545     sloppyline({34,25},{37,13})
546   }
547 }
548 % intentionally not documented:
549 \long\def\drawunicorn{
550   \color{pink!90!black}
551   \drawhorse
552   \luadraw{0}{
553     sloppyline({15,20},{15,50})
554     sloppyline({15,50},{25,20})
555   }
556 }
557 \long\def\drawfatunicorn{

```

```

558 \color{pink!90!black}
559 \drawfathorse
560 \luadraw{0}{
561     sloppyline({15,20},{15,50})
562     sloppyline({15,50},{25,20})
563 }
564 }

```

10 L^AT_EX package

I have decided to keep the L^AT_EX-part of this package as small as possible. So far, it does ... nothing useful, but it provides a `chickenize.sty` that loads `chickenize.tex` so the user can still say `\usepackage{chickenize}`. This file will never support package options!

Some code might be implemented to manipulate figures for full chickenization. However, I will *not* load any packages at this place, as loading of `expl3` or `TikZ` or whatever takes too much time for such a tiny package like this one. If you require any of the features presented here, you have to load the packages on your own. Maybe this will change.

```

565 \ProvidesPackage{chickenize}%
566 [2021/07/21 v0.3a chickenize package]
567 \input{chickenize}

```

10.1 Free Compliments

```

568 %

```

10.2 Definition of User-Level Macros

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

```

569 \iffalse
570 \DeclareDocumentCommand\includegraphics{0}{m}{
571     \fbox{Chicken} %% actually, I'd love to draw an MP graph showing a chicken ...
572 }
573 %%% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
574 %% So far, you have to load pgfplots yourself.
575 %% As it is a mighty package, I don't want the user to force loading it.
576 \NewDocumentCommand\balmerpeak{G{}0{-4cm}}{
577 %% to be done using Lua drawing.
578 }
579 \fi

```

11 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.

```
580 local nodeid    = node.id
581 local nodecopy  = node.copy
582 local nodenew   = node.new
583 local nodetail  = node.tail
584 local nodeslide = node.slide
585 local noderemove = node.remove
586 local nodetraverse = node.traverse
587 local nodetraverseid = node.traverse_id
588 local nodeinsertafter = node.insert_after
589 local nodeinsertbefore = node.insert_before
590
591 Hhead = nodeid("hhead")
592 RULE  = nodeid("rule")
593 GLUE  = nodeid("glue")
594 WHAT  = nodeid("whatsit")
595 COL   = node.subtype("pdf_colorstack")
596 DISC  = nodeid("disc")
597 GLYPH = nodeid("glyph")
598 GLUE  = nodeid("glue")
599 HLIST = nodeid("hlist")
600 KERN  = nodeid("kern")
601 PUNCT = nodeid("punct")
602 PENALTY = nodeid("penalty")
603 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.

```
604 color_push = nodenew(WHAT, COL)
605 color_pop  = nodenew(WHAT, COL)
606 color_push.stack = 0
607 color_pop.stack = 0
608 color_push.command = 1
609 color_pop.command = 2
```

11.1 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.

```
610 chicken_pagenumbers = true
611
612 chickenstring = {}
613 chickenstring[1] = "chicken" -- chickenstring is a table, please remeber this!
614
615 chickenizefraction = 0.5 -- set this to a small value to fool somebody,
```

```

616 -- or to see if your text has been read carefully. This is also a great way to lay easter eggs for
617 chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing your
618
619 local match = unicode.utf8.match
620 chickenize_ignore_word = false
The function chickenize_real_stuff is started once the beginning of a to-be-substituted word is found.
621 chickenize_real_stuff = function(i,head)
622     while ((i.next.id == GLYPH) or (i.next.id == KERN) or (i.next.id == DISC) or (i.next.id == HL
        find end of a word
623         i.next = i.next.next
624     end
625
626     chicken = {} -- constructing the node list.
627
628 -- Should this be done only once? No, otherwise we lose the freedom to change the string in-
        document.
629 -- But it could be done only once each paragraph as in-paragraph changes are not possible!
630
631     chickenstring_tmp = chickenstring[math.random(1,#chickenstring)]
632     chicken[0] = nodenew(GLYPH,1) -- only a dummy for the loop
633     for i = 1,string.len(chickenstring_tmp) do
634         chicken[i] = nodenew(GLYPH,1)
635         chicken[i].font = font.current()
636         chicken[i-1].next = chicken[i]
637     end
638
639     j = 1
640     for s in string.utfvalues(chickenstring_tmp) do
641         local char = unicode.utf8.char(s)
642         chicken[j].char = s
643         if match(char,"%s") then
644             chicken[j] = nodenew(GLUE)
645             chicken[j].width = space
646             chicken[j].shrink = shrink
647             chicken[j].stretch = stretch
648         end
649         j = j+1
650     end
651
652     nodeslide(chicken[1])
653     lang.hyphenate(chicken[1])
654     chicken[1] = node.kerning(chicken[1]) -- FIXME: does not work
655     chicken[1] = node.ligaturing(chicken[1]) -- dito
656
657     nodeinsertbefore(head,i,chicken[1])
658     chicken[1].next = chicken[2] -- seems to be necessary ... to be fixed

```

```

659     chicken[string.len(chickenstring_tmp)].next = i.next
660
661     -- shift lowercase latin letter to uppercase if the original input was an uppercase
662     if (chickenize_capital and (chicken[1].char > 96 and chicken[1].char < 123)) then
663         chicken[1].char = chicken[1].char - 32
664     end
665
666     return head
667 end
668
669 chickenize = function(head)
670     for i in nodetraverseid(GLYPH,head) do --find start of a word
671         -- Random determination of the chickenization of the next word:
672         if math.random() > chickenizefraction then
673             chickenize_ignore_word = true
674         elseif chickencount then
675             chicken_substitutions = chicken_substitutions + 1
676         end
677
678         if (chickenize_ignore_word == false) then -- normal case: at the beginning of a word, we jump
679             if (i.char > 64 and i.char < 91) then chickenize_capital = true else chickenize_capital = false
680             head = chickenize_real_stuff(i,head)
681         end
682
683         -- At the end of the word, the ignoring is reset. New chance for everyone.
684         if not((i.next.id == GLYPH) or (i.next.id == DISC) or (i.next.id == PUNCT) or (i.next.id == KERN)) then
685             chickenize_ignore_word = false
686         end
687     end
688     return head
689 end
690

```

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)

```

691 local separator      = string.rep("=", 28)
692 local texiowrite_nl = texio.write_nl
693 nicetext = function()
694     texiowrite_nl("Output written on "..tex.jobname.."pdf ("..status.total_pages.." chicken,".." eg
695     texiowrite_nl(" ")
696     texiowrite_nl(separator)
697     texiowrite_nl("Hello my dear user,")
698     texiowrite_nl("good job, now go outside and enjoy the world!")
699     texiowrite_nl(" ")
700     texiowrite_nl("And don't forget to feed your chicken!")
701     texiowrite_nl(separator .. "\n")
702     if chickencount then

```



```

703     texiowrite_nl("There were "..chicken_substitutions.." substitutions made.")
704     texiowrite_nl(separator)
705 end
706 end

```

11.2 shownodes

This function is intended for learning and debugging. It will output the id of nodes. You can specify which chain of nodes you want to investigate, and it will try to show all nodes in that chain – e. g. in a paragraph, in math, etc. Examples will be given somewhere.

```

707
708 printnodes = function(head)
709   for i in nodetraverse(head) do
710     texio.write_nl(i.id)
711     if i.id == 0 then
712       printnodes(i.head)
713     end
714   end
715   return head
716 end
717
718 shownodes = function(head)
719 -- start assuming we are in post_linebreak_filter for simplicity
720 -- then we need a function that goes through all hlists recursively
721   printnodes(head)
722
723 --[[
724   if (shownodes_var == "pre_linebreak_filter") then
725     texio.write_nl("-start par-")
726     for i in nodetraverse(head) do
727       texio.write_nl(i.id)
728       if i.id == 0 then
729         texio.write_nl("yo")
730         for i in nodetraverse(i.head) do
731           texio.write_nl("yo" .. i.id)
732         end
733         texio.write_nl("bye")
734       end
735     end
736     texio.write_nl("-end-\n")
737   end
738   if shownodes_var == "post_linebreak_filter" then
739     texio.write_nl("-start par-")
740     for l in nodetraverse(head) do
741       texio.write_nl("-start line-")
742       for i in nodetraverse(l.head) do

```

```

743         texio.write_nl(i.id)
744     end
745     texio.write_nl("--end line-")
746 end
747 end--]]
748 return head
749 end

```

11.3 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:

```

750 boustrophedon = function(head)
751   rot = node.new(WHAT,PDF_LITERAL)
752   rot2 = node.new(WHAT,PDF_LITERAL)
753   odd = true
754   for line in node.traverse_id(0,head) do
755     if odd == false then
756       w = line.width/65536*0.99625 -- empirical correction factor (?)
757       rot.data = "-1 0 0 1 \"..w..\" 0 cm"
758       rot2.data = "-1 0 0 1 \"..-w..\" 0 cm"
759       line.head = node.insert_before(line.head,line.head,nodecopy(rot))
760       nodeinsertafter(line.head,nodetail(line.head),nodecopy(rot2))
761       odd = true
762     else
763       odd = false
764     end
765   end
766   return head
767 end

```

Glyphwise rotation:

```

768 boustrophedon_glyphs = function(head)
769   odd = false
770   rot = nodenew(WHAT,PDF_LITERAL)
771   rot2 = nodenew(WHAT,PDF_LITERAL)
772   for line in nodetraverseid(0,head) do
773     if odd==true then
774       line.dir = "TRT"
775       for g in nodetraverseid(GLYPH,line.head) do
776         w = -g.width/65536*0.99625
777         rot.data = "-1 0 0 1 \" .. w ..\" 0 cm"
778         rot2.data = "-1 0 0 1 \" .. -w ..\" 0 cm"
779         line.head = node.insert_before(line.head,g,nodecopy(rot))
780         nodeinsertafter(line.head,g,nodecopy(rot2))

```

```

781     end
782     odd = false
783     else
784         line.dir = "TLT"
785         odd = true
786     end
787 end
788 return head
789 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.

```

790 boustrophedon_inverse = function(head)
791     rot = node.new(WHAT,PDF_LITERAL)
792     rot2 = node.new(WHAT,PDF_LITERAL)
793     odd = true
794     for line in node.traverse_id(0,head) do
795         if odd == false then
796 texio.write_nl(line.height)
797             w = line.width/65536*0.99625 -- empirical correction factor (?)
798             h = line.height/65536*0.99625
799             rot.data = "-1 0 0 -1 "..w.." "..h.." cm"
800             rot2.data = "-1 0 0 -1"..-w.." "..0.5*h.." cm"
801             line.head = node.insert_before(line.head,line.head,node.copy(rot))
802             node.insert_after(line.head,node.tail(line.head),node.copy(rot2))
803             odd = true
804         else
805             odd = false
806         end
807     end
808     return head
809 end

```

11.4 bubblesort

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

```

810 function bubblesort(head)
811     for line in nodetraverseid(0,head) do
812         for glyph in nodetraverseid(GLYPH,line.head) do
813
814             end
815         end
816     return head
817 end

```

11.5 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!

```
818 countglyphs = function(head)
819   for line in nodetraverseid(0,head) do
820     for glyph in nodetraverseid(GLYPH,line.head) do
821       glyphnumber = glyphnumber + 1
822       if (glyph.next.next) then
823         if (glyph.next.id == 10) and (glyph.next.next.id == GLYPH) then
824           spacenumber = spacenumber + 1
825         end
826         counted_glyphs_by_code[glyph.char] = counted_glyphs_by_code[glyph.char] + 1
827       end
828     end
829   end
830   return head
831 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? ...

```
832 printglyphnumber = function()
833   texiowrite_nl("\nNumber of glyphs by character code (only up to 127):")
834   for i = 1,127 do --%% FIXME: should allow for more characters, but cannot be printed to console
835     texiowrite_nl(string.char(i)..": " ..counted_glyphs_by_code[i])
836   end
837
838   texiowrite_nl("\nTotal number of glyphs in this document: " ..glyphnumber)
839   texiowrite_nl("Number of spaces in this document: " ..spacenumber)
840   texiowrite_nl("Glyphs plus spaces: " ..glyphnumber+spacenumber.." \n")
841 end
```

11.6 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.

```
842 countwords = function(head)
843   for glyph in nodetraverseid(GLYPH,head) do
844     if (glyph.next.id == GLUE) then
845       wordnumber = wordnumber + 1
846     end
847   end
848   wordnumber = wordnumber + 1 -- add 1 for the last word in a paragraph which is not found otherwise
849   return head
850 end
```

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

```
851 printwordnumber = function()
852   texiowrite_nl("\nNumber of words in this document: "..wordnumber)
853 end
```

11.7 detectdoublewords

```
854 %% FIXME: Does this work? ...
855 detectdoublewords = function(head)
856   prevlastword = {} -- array of numbers representing the glyphs
857   prevfirstword = {}
858   newlastword = {}
859   newfirstword = {}
860   for line in nodetraverseid(0,head) do
861     for g in nodetraverseid(GLYPH,line.head) do
862       texio.write_nl("next glyph",#newfirstword+1)
863       newfirstword[#newfirstword+1] = g.char
864       if (g.next.id == 10) then break end
865     end
866     texio.write_nl("nfw:"..#newfirstword)
867   end
868 end
869
870 printdoublewords = function()
871   texio.write_nl("finished")
872 end
```

11.8 francize

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

```
873 francize = function(head)
874   for n in nodetraverseid(GLYPH,head) do
875     if ((n.char > 47) and (n.char < 58)) then
876       n.char = math.random(48,57)
877     end
878   end
879 end
```

```

878 end
879 return head
880 end

```

11.9 gameofchicken

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 \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 $\text{Lua}\TeX$ 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.

```

881 function gameofchicken()
882   GOC_lifetab = {}
883   GOC_spawntab = {}
884   GOC_antilifetab = {}
885   GOC_antispawntab = {}
886   -- translate the rules into an easily-manageable table
887   for i=1,#GOCrule_live do; GOC_lifetab[GOCrule_live[i]] = true end
888   for i=1,#GOCrule_spawn do; GOC_spawntab[GOCrule_spawn[i]] = true end
889   for i=1,#GOCrule_antilive do; GOC_antilifetab[GOCrule_antilive[i]] = true end
890   for i=1,#GOCrule_antispawn do; GOC_antispawntab[GOCrule_antispawn[i]] = true end
891   -- initialize the arrays
892   local life = {}
893   local antilife = {}
894   local newlife = {}
895   local newantilife = {}
896   for i = 0, GOCx do life[i] = {}; newlife[i] = {} for j = 0, GOCy do life[i][j] = 0 end end
897   for i = 0, GOCx do antilife[i] = {}; newantilife[i] = {} for j = 0, GOCy do antilife[i][j] = 0 end end
898   -- These are the functions doing the actual work, checking the neighbors and applying the rules defined above.
899   function applyruleslife(neighbors, lifeij, antineighbors, antilifeij)
900     if GOC_spawntab[neighbors] then myret = 1 else -- new cell
901       if GOC_lifetab[neighbors] and (lifeij == 1) then myret = 1 else myret = 0 end
902     end
903     if antineighbors > 1 then myret = 0 end
904     return myret
905   end
906   function applyrulesantilife(neighbors, lifeij, antineighbors, antilifeij)
907     if (antineighbors == 3) then myret = 1 else -- new cell or keep cell
908       if (((antineighbors > 1) and (antineighbors < 4)) and (lifeij == 1)) then myret = 1 else myret = 0 end
909     end
910     if neighbors > 1 then myret = 0 end
911     return myret
912   end
913 end

```

Preparing the initial state with a default pattern:

```

910 -- prepare some special patterns as starter

```

```
911 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:

```
912 print("start");
913 for i = 1,GOCx do
914     for j = 1,GOCy do
915         if (life[i][j]==1) then texio.write("X") else if (antilife[i][j]==1) then texio.write("0")
916     end
917     texio.write_nl(" ");
918 end
919 os.sleep(GOCsleep)
920
921 for i = 0, GOCx do
922     for j = 0, GOCy do
923         newlife[i][j] = 0 -- Fill the values from the start settings here
924         newantilife[i][j] = 0 -- Fill the values from the start settings here
925     end
926 end
927
928 for k = 1,GOCiter do -- iterate over the cycles
929     texio.write_nl(k);
930     for i = 1, GOCx-1 do -- iterate over lines
931         for j = 1, GOCy-1 do -- iterate over columns -- prevent edge effects
932             local neighbors = (life[i-1][j-1] + life[i-1][j] + life[i-1][j+1] + life[i][j-
933             1] + life[i][j+1] + life[i+1][j-1] + life[i+1][j] + life[i+1][j+1])
934             local antineighbors = (antilife[i-1][j-1] + antilife[i-1][j] + antilife[i-
935             1][j+1] + antilife[i][j-1] + antilife[i][j+1] + antilife[i+1][j-1] + antilife[i+1][j] + antilife[i+1][j+1])
936             newlife[i][j] = applyruleslife(neighbors, life[i][j],antineighbors, antilife[i][j])
937             newantilife[i][j] = applyrulesantilife(neighbors,life[i][j], antineighbors,antilife[i][j])
938         end
939     end
940     for i = 1, GOCx do
941         for j = 1, GOCy do
942             life[i][j] = newlife[i][j] -- copy the values
943             antilife[i][j] = newantilife[i][j] -- copy the values
944         end
945     end
946
947     for i = 1,GOCx do
948         for j = 1,GOCy do
949             if GOC_console then
950                 if (life[i][j]==1) then texio.write("X") else if (antilife[i][j]==1) then texio.write("0")
951             end
952             if GOC_pdf then
953                 if (life[i][j]==1) then tex.print("\placeat("..(i/10)..","..(j/10).."){"..GOCcellcode..
```

```

954         if (antilife[i][j]==1) then tex.print("\\placeat("..(i/10)..","..(j/10).."){"..GOCantic
955     end
956 end
957 end
958 tex.print("\\newpage")
959 os.sleep(GOCsleep)
960 end
961 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.

```

962 function make_a_gif()
963   os.execute("convert -verbose -dispose previous -background white -alpha remove -
      alpha off -density "..GOCdensity.." "..tex.jobname ..".pdf " ..tex.jobname.."gif")
964   os.execute("gwenview "..tex.jobname.."gif")
965 end

```

11.10 guttenbergenize

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

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

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

11.10.1 guttenbergenize – preliminaries

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

```

966 local quotestrings = {
967   [171] = true, [172] = true,
968   [8216] = true, [8217] = true, [8218] = true,
969   [8219] = true, [8220] = true, [8221] = true,
970   [8222] = true, [8223] = true,
971   [8248] = true, [8249] = true, [8250] = true,
972 }

```

11.10.2 guttenbergenize – the function

```

973 guttenbergenize_rq = function(head)
974   for n in nodetraverseid(GLYPH,head) do

```

¹¹Thanks to Jasper for bringing me to this idea!


```

975     local i = n.char
976     if quotestrings[i] then
977         noderemove(head,n)
978     end
979 end
980 return head
981 end

```

11.11 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.¹²

```

982 hammertimedelay = 1.2
983 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
984 hammertime = function(head)
985     if hammerfirst then
986         texiowrite_nl(htime_separator)
987         texiowrite_nl("=====STOP!=====\\n")
988         texiowrite_nl(htime_separator .. "\\n\\n\\n")
989         os.sleep (hammertimedelay*1.5)
990         texiowrite_nl(htime_separator .. "\\n")
991         texiowrite_nl("=====HAMMERTIME=====\\n")
992         texiowrite_nl(htime_separator .. "\\n\\n")
993         os.sleep (hammertimedelay)
994         hammerfirst = false
995     else
996         os.sleep (hammertimedelay)
997         texiowrite_nl(htime_separator)
998         texiowrite_nl("=====U can't touch this!=====\\n")
999         texiowrite_nl(htime_separator .. "\\n\\n")
1000         os.sleep (hammertimedelay*0.5)
1001     end
1002     return head
1003 end

```

11.12 italianize

This is inspired by some of the more melodic pronounciations 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 ad randomly an e to words ending in consonants. This is tricky and might fail – I’m happy to receive and try to solve ayn bug reports.

```

1004 italianizefraction = 0.5 --%% gives the amount of italianization
1005 mynode = nodenew(GLYPH) -- prepare a dummy glyph

```

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

```

1006
1007 italianize = function(head)
1008   -- skip "h/H" randomly
1009   for n in node.traverse_id(GLYPH,head) do -- go through all glyphs
1010     if n.prev.id ~= GLYPH then -- check if it's a word start
1011       if ((n.char == 72) or (n.char == 104)) and (tex.normal_rand() < italianizefraction) then --
1012         n.prev.next = n.next
1013       end
1014     end
1015   end
1016
1017   -- add h or H in front of vowels
1018   for n in node.traverse_id(GLYPH,head) do
1019     if math.random() < italianizefraction then
1020       x = n.char
1021       if x == 97 or x == 101 or x == 105 or x == 111 or x == 117 or
1022         x == 65 or x == 69 or x == 73 or x == 79 or x == 85 then
1023         if (n.prev.id == GLUE) then
1024           mynode.font = n.font
1025           if x > 90 then -- lower case
1026             mynode.char = 104
1027           else
1028             mynode.char = 72 -- upper case - convert into lower case
1029             n.char = x + 32
1030           end
1031           node.insert_before(head,n,node.copy(mynode))
1032         end
1033       end
1034     end
1035   end
1036
1037   -- add e after words, but only after consonants
1038   for n in node.traverse_id(GLUE,head) do
1039     if n.prev.id == GLYPH then
1040       x = n.prev.char
1041       -- skip vowels and randomize
1042       if not(x == 97 or x == 101 or x == 105 or x == 111 or x == 117 or x == 44 or x == 46) and math.random() < italianizefraction then
1043         mynode.char = 101 -- it's always a lower case e, no?
1044         mynode.font = n.prev.font -- adapt the current font
1045         node.insert_before(head,n,node.copy(mynode)) -- insert the e in the node list
1046       end
1047     end
1048   end
1049
1050   return head
1051 end

```

11.13 italianizerandwords

This is inspired by my dearest colleagues and their artistic interpretation of the english grammar. The command will cause LuaTeX to read a sentence (i. e. text until the next full stop), then randomizes the words (i. e. units separated by a space) in it and throws the result back to the typesetting. Useless? Very.

```
1052 italianizerandwords = function(head)
1053 words = {}
1054 wordnumber = 0
1055 -- head.next.next is the very first word. However, let's try to get the first word after the first
1056 for n in nodetraverseid(GLUE,head) do -- let's try to count words by their separators
1057     wordnumber = wordnumber + 1
1058     if n.next then
1059         words[wordnumber] = {}
1060         words[wordnumber][1] = node.copy(n.next)
1061
1062         glyphnumber = 1
1063         myglyph = n.next
1064         while myglyph.next do
1065             node.tail(words[wordnumber][1]).next = node.copy(myglyph.next)
1066             myglyph = myglyph.next
1067         end
1068     end
1069 print(#words)
1070 if #words > 0 then
1071     print("lengs is: ")
1072     print(#words[#words])
1073 end
1074 end
1075 --myinsertnode = head.next.next -- first letter
1076 --node.tail(words[1][1]).next = myinsertnode.next
1077 --myinsertnode.next = words[1][1]
1078
1079 return head
1080 end
1081
1082 italianize_old = function(head)
1083     local wordlist = {} -- here we will store the number of words of the sentence.
1084     local words = {} -- here we will store the words of the sentence.
1085     local wordnumber = 0
1086     -- let's first count all words in one sentence, howboutdat?
1087     wordlist[wordnumber] = 1 -- let's save the word *length* in here ...
1088
1089
1090 for n in nodetraverseid(GLYPH,head) do
1091     if (n.next.id == GLUE) then -- this is a space
1092         wordnumber = wordnumber + 1
```

```

1093     wordlist[wordnumber] = 1
1094     words[wordnumber] = n.next.next
1095 end
1096 if (n.next.id == GLYPH) then -- it's a glyph
1097 if (n.next.char == 46) then -- this is a full stop.
1098     wordnumber = wordnumber + 1
1099     texio.write_nl("this sentence had "..wordnumber.."words.")
1100     for i=0,wordnumber-1 do
1101         texio.write_nl("word "..i.." had " .. wordlist[i] .. "glyphs")
1102     end
1103     texio.write_nl(" ")
1104     wordnumber = -1 -- to compensate the fact that the next node will be a space, this would co
1105 else
1106
1107     wordlist[wordnumber] = wordlist[wordnumber] + 1 -- the current word got 1 glyph longer
1108 end
1109 end
1110 end
1111 return head
1112 end

```

11.14 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.

```

1113 itsame = function()
1114 local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
1115 color = "1 .6 0"
1116 for i = 6,9 do mr(i,3) end
1117 for i = 3,11 do mr(i,4) end
1118 for i = 3,12 do mr(i,5) end
1119 for i = 4,8 do mr(i,6) end
1120 for i = 4,10 do mr(i,7) end
1121 for i = 1,12 do mr(i,11) end
1122 for i = 1,12 do mr(i,12) end
1123 for i = 1,12 do mr(i,13) end
1124
1125 color = ".3 .5 .2"
1126 for i = 3,5 do mr(i,3) end mr(8,3)
1127 mr(2,4) mr(4,4) mr(8,4)
1128 mr(2,5) mr(4,5) mr(5,5) mr(9,5)
1129 mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
1130 for i = 3,8 do mr(i,8) end
1131 for i = 2,11 do mr(i,9) end
1132 for i = 1,12 do mr(i,10) end
1133 mr(3,11) mr(10,11)

```

```

1134 for i = 2,4 do mr(i,15) end for i = 9,11 do mr(i,15) end
1135 for i = 1,4 do mr(i,16) end for i = 9,12 do mr(i,16) end
1136
1137 color = "1 0 0"
1138 for i = 4,9 do mr(i,1) end
1139 for i = 3,12 do mr(i,2) end
1140 for i = 8,10 do mr(5,i) end
1141 for i = 5,8 do mr(i,10) end
1142 mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
1143 for i = 4,9 do mr(i,12) end
1144 for i = 3,10 do mr(i,13) end
1145 for i = 3,5 do mr(i,14) end
1146 for i = 7,10 do mr(i,14) end
1147 end

```

11.15 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.

```

1148 chickenkernamount = 0
1149 chickeninvertkerning = false
1150
1151 function kernmanipulate (head)
1152   if chickeninvertkerning then -- invert the kerning
1153     for n in nodetraverseid(11,head) do
1154       n.kern = -n.kern
1155     end
1156   else -- if not, set it to the given value
1157     for n in nodetraverseid(11,head) do
1158       n.kern = chickenkernamount
1159     end
1160   end
1161   return head
1162 end

```

11.16 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.

```

1163 leetspeak_onlytext = false
1164 leettable = {
1165   [101] = 51, -- E
1166   [105] = 49, -- I

```

```

1167 [108] = 49, -- L
1168 [111] = 48, -- O
1169 [115] = 53, -- S
1170 [116] = 55, -- T
1171
1172 [101-32] = 51, -- e
1173 [105-32] = 49, -- i
1174 [108-32] = 49, -- l
1175 [111-32] = 48, -- o
1176 [115-32] = 53, -- s
1177 [116-32] = 55, -- t
1178 }

```

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

```

1179 leet = function(head)
1180   for line in nodetraverseid(Hhead,head) do
1181     for i in nodetraverseid(GLYPH,line.head) do
1182       if not leetspeak_onlytext or
1183         node.has_attribute(i,luatexbase.attributes.leetattr)
1184       then
1185         if leettable[i.char] then
1186           i.char = leettable[i.char]
1187         end
1188       end
1189     end
1190   end
1191   return head
1192 end

```

11.17 leftsideright

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

```

1193 leftsideright = function(head)
1194   local factor = 65536/0.99626
1195   for n in nodetraverseid(GLYPH,head) do
1196     if (leftsiderightarray[n.char]) then
1197       shift = nodenew(WHAT,PDF_LITERAL)
1198       shift2 = nodenew(WHAT,PDF_LITERAL)
1199       shift.data = "q -1 0 0 1 " .. n.width/factor .. " 0 cm"
1200       shift2.data = "Q 1 0 0 1 " .. n.width/factor .. " 0 cm"
1201       nodeinsertbefore(head,n,shift)
1202       nodeinsertafter(head,n,shift2)
1203     end
1204   end
1205   return head
1206 end

```

11.18 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>

11.18.1 setup of variables

```
1207 local letterspace_glue = nodenew(GLUE)
1208 local letterspace_pen   = nodenew(PENALTY)
1209
1210 letterspace_glue.width   = tex.sp"0pt"
1211 letterspace_glue.stretch = tex.sp"0.5pt"
1212 letterspace_pen.penalty = 10000
```

11.18.2 function implementation

```
1213 letterspaceadjust = function(head)
1214   for glyph in nodetraverseid(GLYPH, head) do
1215     if glyph.prev and (glyph.prev.id == GLYPH or glyph.prev.id == DISC or glyph.prev.id == KERN) then
1216       local g = nodecopy(letterspace_glue)
1217       nodeinsertbefore(head, glyph, g)
1218       nodeinsertbefore(head, g, nodecopy(letterspace_pen))
1219     end
1220   end
1221   return head
1222 end
```

11.18.3 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!

```
1223 textletterspaceadjust = function(head)
1224   for glyph in nodetraverseid(GLYPH, head) do
1225     if node.has_attribute(glyph, luatexbase.attributes.letterspaceadjustattr) then
1226       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
1227         local g = node.copy(letterspace_glue)
1228         nodeinsertbefore(head, glyph, g)
1229         nodeinsertbefore(head, g, nodecopy(letterspace_pen))
1230       end
1231     end
1232   end
1233   luatexbase.remove_from_callback("pre_linebreak_filter", "textletterspaceadjust")
1234   return head
```

1235 end

11.19 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.

```
1236 matrixize = function(head)
1237   x = {}
1238   s = nodenew(DISC)
1239   for n in nodetraverseid(GLYPH,head) do
1240     j = n.char
1241     for m = 0,7 do -- stay ASCII for now
1242       x[7-m] = nodecopy(n) -- to get the same font etc.
1243     end
1244     if (j / (2^(7-m)) < 1) then
1245       x[7-m].char = 48
1246     else
1247       x[7-m].char = 49
1248       j = j-(2^(7-m))
1249     end
1250     nodeinsertbefore(head,n,x[7-m])
1251     nodeinsertafter(head,x[7-m],nodecopy(s))
1252   end
1253   noderemove(head,n)
1254 end
1255 return head
1256 end
```

11.20 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 ...

```
1257 medievalumlaut = function(head)
1258   local factor = 65536/0.99626
1259   local org_e_node = nodenew(GLYPH)
1260   org_e_node.char = 101
1261   for line in nodetraverseid(0,head) do
1262     for n in nodetraverseid(GLYPH,line.head) do
1263       if (n.char == 228 or n.char == 246 or n.char == 252) then
1264         e_node = nodecopy(org_e_node)
1265         e_node.font = n.font
```



```

1266     shift = nodenew(WHAT,PDF_LITERAL)
1267     shift2 = nodenew(WHAT,PDF_LITERAL)
1268     shift2.data = "Q 1 0 0 1 " .. e_node.width/factor .. " 0 cm"
1269     nodeinsertafter(head,n,e_node)
1270
1271     nodeinsertbefore(head,e_node,shift)
1272     nodeinsertafter(head,e_node,shift2)
1273
1274     x_node = nodenew(KERN)
1275     x_node.kern = -e_node.width
1276     nodeinsertafter(head,shift2,x_node)
1277 end
1278
1279 if (n.char == 228) then -- ä
1280     shift.data = "q 0.5 0 0 0.5 " ..
1281         -n.width/factor*0.85 .. " .. n.height/factor*0.75 .. " cm"
1282     n.char = 97
1283 end
1284 if (n.char == 246) then -- ö
1285     shift.data = "q 0.5 0 0 0.5 " ..
1286         -n.width/factor*0.75 .. " .. n.height/factor*0.75 .. " cm"
1287     n.char = 111
1288 end
1289 if (n.char == 252) then -- ü
1290     shift.data = "q 0.5 0 0 0.5 " ..
1291         -n.width/factor*0.75 .. " .. n.height/factor*0.75 .. " cm"
1292     n.char = 117
1293 end
1294 end
1295 end
1296 return head
1297 end

```

11.21 pancakenize

```

1298 local separator      = string.rep("=", 28)
1299 local texiowrite_nl = texio.write_nl
1300 pancaketext = function()
1301     texiowrite_nl("Output written on "..tex.jobname.."pdf ("..status.total_pages.." chicken,".." eg
1302     texiowrite_nl(" ")
1303     texiowrite_nl(separator)
1304     texiowrite_nl("Soo ... you decided to use \\pancakenize.")
1305     texiowrite_nl("That means you owe me a pancake!")
1306     texiowrite_nl(" ")
1307     texiowrite_nl("(This goes by document, not compilation.)")
1308     texiowrite_nl(separator.."\\n\\n")

```

```

1309 texiowrite_nl("Looking forward for my pancake! :)")
1310 texiowrite_nl("\n\n")
1311 end

```

11.22 randomerror

Not yet implemented, sorry.

11.23 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.

```

1312 randomfontslower = 1
1313 randomfontsupper = 0
1314 %
1315 randomfonts = function(head)
1316   local rfub
1317   if randomfontsupper > 0 then -- fixme: this should be done only once, no? Or at every paragraph
1318     rfub = randomfontsupper -- user-specified value
1319   else
1320     rfub = font.max() -- or just take all fonts
1321   end
1322   for line in nodetraverseid(Hhead,head) do
1323     for i in nodetraverseid(GLYPH,line.head) do
1324       if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattrib) then
1325         i.font = math.random(randomfontslower,rfub)
1326       end
1327     end
1328   end
1329   return head
1330 end

```

11.24 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```

1331 uclcratio = 0.5 -- ratio between uppercase and lower case
1332 randomuclc = function(head)
1333   for i in nodetraverseid(GLYPH,head) do
1334     if not(randomuclc_onlytext) or node.has_attribute(i,luatexbase.attributes.randuclcattrib) then
1335       if math.random() < uclcratio then
1336         i.char = tex.uccode[i.char]
1337       else
1338         i.char = tex.lccode[i.char]
1339       end
1340     end
1341   end
1342   return head

```

```
1343 end
```

11.25 randomchars

```
1344 randomchars = function(head)
1345   for line in nodetraverseid(Hhead,head) do
1346     for i in nodetraverseid(GLYPH,line.head) do
1347       i.char = math.floor(math.random()*512)
1348     end
1349   end
1350   return head
1351 end
```

11.26 randomcolor and rainbowcolor

11.26.1 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.

```
1352 randomcolor_grey = false
1353 randomcolor_onlytext = false --switch between local and global colorization
1354 rainbowcolor = false
1355
1356 grey_lower = 0
1357 grey_upper = 900
1358
1359 Rgb_lower = 1
1360 rGb_lower = 1
1361 rgB_lower = 1
1362 Rgb_upper = 254
1363 rGb_upper = 254
1364 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.

```
1365 rainbow_step = 0.005
1366 rainbow_Rgb = 1-rainbow_step -- we start in the red phase
1367 rainbow_rGb = rainbow_step -- values x must always be  $0 < x < 1$ 
1368 rainbow_rgB = rainbow_step
1369 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.

```
1370 randomcolorstring = function()
1371   if randomcolor_grey then
1372     return (0.001*math.random(grey_lower,grey_upper)).." g"
1373   elseif rainbowcolor then
1374     if rainind == 1 then -- red
1375       rainbow_rGb = rainbow_rGb + rainbow_step
```

```

1376     if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
1377 elseif rainind == 2 then -- yellow
1378     rainbow_Rgb = rainbow_Rgb - rainbow_step
1379     if rainbow_Rgb <= rainbow_step then rainind = 3 end
1380 elseif rainind == 3 then -- green
1381     rainbow_rGb = rainbow_rGb + rainbow_step
1382     rainbow_rGb = rainbow_rGb - rainbow_step
1383     if rainbow_rGb <= rainbow_step then rainind = 4 end
1384 elseif rainind == 4 then -- blue
1385     rainbow_Rgb = rainbow_Rgb + rainbow_step
1386     if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
1387 else -- purple
1388     rainbow_rGb = rainbow_rGb - rainbow_step
1389     if rainbow_rGb <= rainbow_step then rainind = 1 end
1390 end
1391 return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rGb.." rg"
1392 else
1393     Rgb = math.random(Rgb_lower,Rgb_upper)/255
1394     rGb = math.random(rGb_lower,rGb_upper)/255
1395     rgB = math.random(rgB_lower,rgB_upper)/255
1396     return Rgb.." "..rGb.." "..rgB.." .." rg"
1397 end
1398 end

```

11.26.2 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.

```

1399 randomcolor = function(head)
1400   for line in nodetraverseid(0,head) do
1401     for i in nodetraverseid(GLYPH,line.head) do
1402 --[[      if not(randomcolor_onlytext) or
1403           (node.has_attribute(i,luatexbase.attributes.randcolorattr))
1404       then--]]
1405         color_push.data = randomcolorstring() -- color or grey string
1406         line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
1407         nodeinsertafter(line.head,i,nodecopy(color_pop))
1408 --      end
1409     end
1410   end
1411   return head
1412 end
1413
1414 pushcolor = function(head)
1415   return head

```

```

1416 end
1417

```

11.27 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.

```

1418 sailheight = 12
1419 mastheight = 4
1420 hullheight = 5
1421 relnumber = 402
1422 function relationship()
1423 --%% check if there's a problem with any character in the current font
1424 f = font.getfont(font.current())
1425 fullfont = 1
1426 for i = 8756,8842 do
1427     if not(f.characters[i]) then texio.write_nl((i).." not available") fullfont = 0 end
1428 end
1429 --%% store the result of the check for later, then go on to construct the ship:
1430 shipheight = sailheight + mastheight + hullheight
1431 tex.print("\parshape " .. (shipheight)) --%% prepare the paragraph shape ...
1432 for i = 1, sailheight do
1433     tex.print(" " .. (4.5 - i / 3.8) .. "cm " .. ((i - 0.5) / 2.5) .. "cm ")
1434 end
1435 for i = 1, mastheight do
1436     tex.print(" " .. (3.2) .. "cm " .. (1) .. "cm ")
1437 end
1438 for i = 1, hullheight do
1439     tex.print(" " .. ((i - 1) / 2) .. "cm " .. (10 - i) .. "cm ")
1440 end
1441 tex.print("\noindent") --%% ... up to here, then insert relations
1442 for i = 1, relnumber do
1443     tex.print("\ \char" .. math.random(8756, 8842))
1444 end
1445 tex.print("\break")
1446 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`.

```

1447 function cutparagraph(head)
1448     local parsum = 0
1449     for n in nodetraverseid(HLIST, head) do
1450         parsum = parsum + 1
1451         if parsum > shipheight then
1452             node.remove(head, n)

```

```

1453     end
1454 end
1455 return head
1456 end

```

And finally a helper function to inform our dear users that they have to use a font that actually can display all the necessary symbols.

```

1457 function missingcharstext()
1458   if (fullfont == 0) then
1459     local separator = string.rep("=", 28)
1460 local texiowrite_nl = texio.write_nl
1461   texiowrite_nl("Output written on "..tex.jobname.."pdf ("..status.total_pages.." chicken,".." e
1462   texiowrite_nl(" ")
1463   texiowrite_nl(separator)
1464   texiowrite_nl("CAREFUL!!")
1465   texiowrite_nl("\\relationship needs special characters (unicode points 8756 to 8842)")
1466   texiowrite_nl("Your font does not support all of them!")
1467   texiowrite_nl("consider using another one, e.g. the XITS font supplied with TeXlive.")
1468   texiowrite_nl(separator .. "\n")
1469   end
1470 end

```

11.28 rickroll

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

```

1471 %

```

11.29 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.

```

1472 substitutewords_strings = {}
1473
1474 addtosubstitutions = function(input,output)
1475   substitutewords_strings[#substitutewords_strings + 1] = {}
1476   substitutewords_strings[#substitutewords_strings][1] = input
1477   substitutewords_strings[#substitutewords_strings][2] = output
1478 end
1479

```

```

1480 substitutewords = function(head)
1481   for i = 1,#substitutewords_strings do
1482     head = string.gsub(head,substitutewords_strings[i][1],substitutewords_strings[i][2])
1483   end
1484   return head
1485 end

```

11.30 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 whether the `next.next` node is also a space. This might not be the best and most universal way of doing it, but the simplest. The penalty is not created newly each time, but copied – no significant speed gain, however.

```

1486 suppressonecharbreakpenaltynode = node.new(PENALTY)
1487 suppressonecharbreakpenaltynode.penalty = 10000
1488 function suppressonecharbreak(head)
1489   for i in node.traverse_id(GLUE,head) do
1490     if ((i.next) and (i.next.next.id == GLUE)) then
1491       pen = node.copy(suppressonecharbreakpenaltynode)
1492       node.insert_after(head,i.next,pen)
1493     end
1494   end
1495
1496   return head
1497 end

```

11.31 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.

```

1498 tabularasa_onlytext = false
1499
1500 tabularasa = function(head)
1501   local s = nodenew(KERN)
1502   for line in nodetraverseid(HLIST,head) do
1503     for n in nodetraverseid(GLYPH,line.head) do
1504       if not(tabularasa_onlytext) or node.has_attribute(n,luatexbase.attributes.tabularasaattr) then
1505         s.kern = n.width
1506         nodeinsertafter(line.list,n,nodecopy(s))
1507         line.head = noderemove(line.list,n)
1508       end
1509     end
1510   end
1511   return head
1512 end

```

11.32 tanjanize

```
1513 tanjanize = function(head)
1514   local s = nodenew(KERN)
1515   local m = nodenew(GLYPH,1)
1516   local use_letter_i = true
1517   scale = nodenew(WHAT,PDF_LITERAL)
1518   scale2 = nodenew(WHAT,PDF_LITERAL)
1519   scale.data = "0.5 0 0 0.5 0 0 cm"
1520   scale2.data = "2 0 0 2 0 0 cm"
1521
1522   for line in nodetraverseid(HLIST,head) do
1523     for n in nodetraverseid(GLYPH,line.head) do
1524       mimicount = 0
1525       tmpwidth = 0
1526       while ((n.next.id == GLYPH) or (n.next.id == 11) or (n.next.id == 7) or (n.next.id == 0)) do
1527         find end of a word
1528         n.next = n.next.next
1529         mimicount = mimicount + 1
1530         tmpwidth = tmpwidth + n.width
1531       end
1532       mimi = {} -- constructing the node list.
1533       mimi[0] = nodenew(GLYPH,1) -- only a dummy for the loop
1534       for i = 1,string.len(mimicount) do
1535         mimi[i] = nodenew(GLYPH,1)
1536         mimi[i].font = font.current()
1537         if(use_letter_i) then mimi[i].char = 109 else mimi[i].char = 105 end
1538         use_letter_i = not(use_letter_i)
1539         mimi[i-1].next = mimi[i]
1540       end
1541     end
1542
1543     line.head = nodeinsertbefore(line.head,n,nodecopy(scale))
1544     nodeinsertafter(line.head,n,nodecopy(scale2))
1545     s.kern = (tmpwidth*2-n.width)
1546     nodeinsertafter(line.head,n,nodecopy(s))
1547   end
1548 end
1549 return head
1550 end
```

11.33 uppercasecolor

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

```
1551 uppercasecolor_onlytext = false
1552
```



```

1553 uppercasecolor = function (head)
1554   for line in nodetraverseid(Hhead,head) do
1555     for upper in nodetraverseid(GLYPH,line.head) do
1556       if not(uppercasecolor_onlytext) or node.has_attribute(upper,luatexbase.attributes.uppercase
1557         if (((upper.char > 64) and (upper.char < 91)) or
1558           ((upper.char > 57424) and (upper.char < 57451))) then -- for small caps! nice
1559           color_push.data = randomcolorstring() -- color or grey string
1560           line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
1561           nodeinsertafter(line.head,upper,nodecopy(color_pop))
1562         end
1563       end
1564     end
1565   end
1566   return head
1567 end

```

11.34 upsidedown

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

```

1568 upsidedown = function(head)
1569   local factor = 65536/0.99626
1570   for line in nodetraverseid(Hhead,head) do
1571     for n in nodetraverseid(GLYPH,line.head) do
1572       if (upsidedownarray[n.char]) then
1573         shift = nodenew(WHAT,PDF_LITERAL)
1574         shift2 = nodenew(WHAT,PDF_LITERAL)
1575         shift.data = "q 1 0 0 -1 0 " .. n.height/factor .. " cm"
1576         shift2.data = "Q 1 0 0 1 " .. n.width/factor .. " 0 cm"
1577         nodeinsertbefore(head,n,shift)
1578         nodeinsertafter(head,n,shift2)
1579       end
1580     end
1581   end
1582   return head
1583 end

```

11.35 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 \LaTeX . The box color then corresponds to the amount of font expansion in the line. This works great for demonstrating the positive effect of font expansion on the badness of a line!

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

11.35.1 colorstretch – preliminaries

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

```
1584 keeptext = true
1585 colorexansion = true
1586
1587 colorstretch_coloroffset = 0.5
1588 colorstretch_colorrange = 0.5
1589 chickenize_rule_bad_height = 4/5 -- height and depth of the rules
1590 chickenize_rule_bad_depth = 1/5
1591
1592
1593 colorstretchnumbers = true
1594 drawstretchthreshold = 0.1
1595 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.

```
1596 colorstretch = function (head)
1597   local f = font.getfont(font.current()).characters
1598   for line in nodetraverseid(Hhead,head) do
1599     local rule_bad = nodenew(RULE)
1600
1601     if colorexansion then -- if also the font expansion should be shown
1602       --%% here use first_glyph function!!
1603       local g = line.head
1604       n = node.first_glyph(line.head.next)
1605       texio.write_nl(line.head.id)
1606       texio.write_nl(line.head.next.id)
1607       texio.write_nl(line.head.next.next.id)
1608       texio.write_nl(n.id)
1609       while not(g.id == GLYPH) and (g.next) do g = g.next end -- find first glyph on line. If line
1610       if (g.id == GLYPH) then -- read width only if g is a glyph!
1611         exp_factor = g.expansion_factor/10000 --%% neato, luatex now directly gives me this!!
1612         exp_color = colorstretch_coloroffset + (exp_factor*0.1) .. " g"
1613       texio.write_nl(exp_factor)
1614       rule_bad.width = 0.5*line.width -- we need two rules on each line!
1615     end
1616   else
1617     rule_bad.width = line.width -- only the space expansion should be shown, only one rule
1618   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.

```
1619 rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bet
1620 rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth
1621
1622 local glue_ratio = 0
1623 if line.glue_order == 0 then
1624   if line.glue_sign == 1 then
1625     glue_ratio = colorstretch_colorrange * math.min(line.glue_set,1)
1626   else
1627     glue_ratio = -colorstretch_colorrange * math.min(line.glue_set,1)
1628   end
1629 end
1630 color_push.data = colorstretch_coloroffset + glue_ratio .. " g"
1631
```

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

```
1632 -- set up output
1633 local p = line.head
1634
1635 -- a rule to immitate kerning all the way back
1636 local kern_back = nodenew(RULE)
1637 kern_back.width = -line.width
1638
1639 -- if the text should still be displayed, the color and box nodes are inserted additionally
1640 -- and the head is set to the color node
1641 if kepttext then
1642   line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1643 else
1644   node.flush_list(p)
1645   line.head = nodecopy(color_push)
1646 end
1647 nodeinsertafter(line.head,line.head,rule_bad) -- then the rule
1648 nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
1649 tmpnode = nodeinsertafter(line.head,line.head.next.next,kern_back)
1650
1651 -- then a rule with the expansion color
1652 if colorexansion then -- if also the stretch/shrink of letters should be shown
1653   color_push.data = exp_color
1654   nodeinsertafter(line.head,tmpnode,nodecopy(color_push))
1655   nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
1656   nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
1657 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.

```
1658   if colorstretchnumbers then
1659       j = 1
1660       glue_ratio_output = {}
1661       for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
1662           local char = unicode.utf8.char(s)
1663           glue_ratio_output[j] = nodenew(GLYPH,1)
1664           glue_ratio_output[j].font = font.current()
1665           glue_ratio_output[j].char = s
1666           j = j+1
1667       end
1668       if math.abs(glue_ratio) > drawstretchthreshold then
1669           if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
1670           else color_push.data = "0 0.99 0 rg" end
1671       else color_push.data = "0 0 0 rg"
1672       end
1673
1674       nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
1675       for i = 1,math.min(j-1,7) do
1676           nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])
1677       end
1678       nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
1679   end -- end of stretch number insertion
1680 end
1681 return head
1682 end
```

scorpionize

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

```
1683 function scorpionize_color(head)
1684   color_push.data = ".35 .55 .75 rg"
1685   nodeinsertafter(head,head,nodecopy(color_push))
1686   nodeinsertafter(head,node.tail(head),nodecopy(color_pop))
1687   return head
1688 end
```

11.36 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 ... (?)

```
1689 substlist = {}
1690 substlist[1488] = 64289
```

```

1691 substlist[1491] = 64290
1692 substlist[1492] = 64291
1693 substlist[1499] = 64292
1694 substlist[1500] = 64293
1695 substlist[1501] = 64294
1696 substlist[1512] = 64295
1697 substlist[1514] = 64296

```

In the function, we need reproducible 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”).

```

1698 function variantjustification(head)
1699   math.randomseed(1)
1700   for line in nodetraverseid(Hhead,head) do
1701     if (line.glue_sign == 1 and line.glue_order == 0) then -- exclude the last line!
1702       substitutions_wide = {} -- we store all "expandable" letters of each line
1703       for n in nodetraverseid(GLYPH,line.head) do
1704         if (substlist[n.char]) then
1705           substitutions_wide[#substitutions_wide+1] = n
1706         end
1707       end
1708       line.glue_set = 0 -- deactivate normal glue expansion
1709       local width = node.dimensions(line.head) -- check the new width of the line
1710       local goal = line.width
1711       while (width < goal and #substitutions_wide > 0) do
1712         x = math.random(#substitutions_wide) -- choose randomly a glyph to be substituted
1713         oldchar = substitutions_wide[x].char
1714         substitutions_wide[x].char = substlist[substitutions_wide[x].char] -- substitute by wide
1715         width = node.dimensions(line.head) -- check if the line is too wide
1716         if width > goal then substitutions_wide[x].char = oldchar break end -- substitute back if
1717         table.remove(substitutions_wide,x) -- if further substitutions have to be done,
1718       end
1719     end
1720   end
1721   return head
1722 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.

11.37 zebrianize

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`.

11.37.1 zebranize – preliminaries

```

1723 zebracolorarray = {}
1724 zebracolorarray_bg = {}
1725 zebracolorarray[1] = "0.1 g"
1726 zebracolorarray[2] = "0.9 g"
1727 zebracolorarray_bg[1] = "0.9 g"
1728 zebracolorarray_bg[2] = "0.1 g"

```

11.37.2 zebranize – the function

This code has to be revisited, it is ugly.

```

1729 function zebranize(head)
1730   zebracolor = 1
1731   for line in nodetraverseid(Hhead,head) do
1732     if zebracolor == #zebracolorarray then zebracolor = 0 end
1733     zebracolor = zebracolor + 1
1734     color_push.data = zebracolorarray[zebracolor]
1735     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1736     for n in nodetraverseid(GLYPH,line.head) do
1737       if n.next then else
1738         nodeinsertafter(line.head,n,nodecopy(color_pull))
1739       end
1740     end
1741
1742     local rule_zebra = nodenew(RULE)
1743     rule_zebra.width = line.width
1744     rule_zebra.height = tex.baselineskip.width*4/5
1745     rule_zebra.depth = tex.baselineskip.width*1/5
1746
1747     local kern_back = nodenew(RULE)
1748     kern_back.width = -line.width
1749
1750     color_push.data = zebracolorarray_bg[zebracolor]
1751     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
1752     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1753     nodeinsertafter(line.head,line.head,kern_back)
1754     nodeinsertafter(line.head,line.head,rule_zebra)
1755   end
1756   return (head)
1757 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 ☺

12 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 `sloppinesssh` 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

```

1758 --
1759 function pdf_print (...)
1760   for _, str in ipairs({...}) do
1761     pdf.print(str .. " ")
1762   end
1763   pdf.print("\n")
1764 end
1765
1766 function move (p1,p2)
1767   if (p2) then
1768     pdf_print(p1,p2,"m")
1769   else
1770     pdf_print(p1[1],p1[2],"m")
1771   end
1772 end
1773
1774 function line(p1,p2)
1775   if (p2) then
1776     pdf_print(p1,p2,"l")
1777   else
1778     pdf_print(p1[1],p1[2],"l")
1779   end
1780 end
1781
1782 function curve(p11,p12,p21,p22,p31,p32)
1783   if (p22) then
1784     p1,p2,p3 = {p11,p12},{p21,p22},{p31,p32}
1785   else
1786     p1,p2,p3 = p11,p12,p21
1787   end
1788   pdf_print(p1[1], p1[2],
1789             p2[1], p2[2],

```



```

1790             p3[1], p3[2], "c")
1791 end
1792
1793 function close ()
1794     pdf_print("h")
1795 end
1796

```

By setting drawwidth to something different than 1 you can adjust the thickness of the strokes. Any stroke done with the sloppy functions will be varied between 0.5 drawwidth and 1.5 drawwidth.

```

1797 drawwidth = 1
1798
1799 function linewidth (w)
1800     pdf_print(w,"w")
1801 end
1802
1803 function stroke ()
1804     pdf_print("S")
1805 end
1806 --
1807
1808 function strictcircle(center,radius)
1809     local left = {center[1] - radius, center[2]}
1810     local lefttop = {left[1], left[2] + 1.45*radius}
1811     local leftbot = {left[1], left[2] - 1.45*radius}
1812     local right = {center[1] + radius, center[2]}
1813     local righttop = {right[1], right[2] + 1.45*radius}
1814     local rightbot = {right[1], right[2] - 1.45*radius}
1815
1816     move (left)
1817     curve (lefttop, righttop, right)
1818     curve (rightbot, leftbot, left)
1819 stroke()
1820 end
1821
1822 sloppynessh = 5
1823 sloppynessv = 5
1824
1825 function disturb_point(point)
1826     return {point[1] + (math.random() - 1/2)*sloppynessh,
1827             point[2] + (math.random() - 1/2)*sloppynessv}
1828 end
1829
1830 function sloppycircle(center,radius)
1831     local left = disturb_point({center[1] - radius, center[2]})
1832     local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
1833     local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}

```

```

1834 local right = disturb_point({center[1] + radius, center[2]})
1835 local righttop = disturb_point({right[1], right[2] + 1.45*radius})
1836 local rightbot = disturb_point({right[1], right[2] - 1.45*radius})
1837
1838 local right_end = disturb_point(right)
1839
1840 move (right)
1841 curve (rightbot, leftbot, left)
1842 curve (lefttop, righttop, right_end)
1843 linewidth(drawwidth*(math.random()+0.5))
1844 stroke()
1845 end
1846
1847 function sloppyellipsis(center,radiusx,radiusy)
1848 local left = disturb_point({center[1] - radiusx, center[2]})
1849 local lefttop = disturb_point({left[1], left[2] + 1.45*radiusy})
1850 local leftbot = {lefttop[1], lefttop[2] - 2.9*radiusy}
1851 local right = disturb_point({center[1] + radiusx, center[2]})
1852 local righttop = disturb_point({right[1], right[2] + 1.45*radiusy})
1853 local rightbot = disturb_point({right[1], right[2] - 1.45*radiusy})
1854
1855 local right_end = disturb_point(right)
1856
1857 move (right)
1858 curve (rightbot, leftbot, left)
1859 curve (lefttop, righttop, right_end)
1860 linewidth(drawwidth*(math.random()+0.5))
1861 stroke()
1862 end
1863
1864 function sloppyline(start,stop)
1865 local start_line = disturb_point(start)
1866 local stop_line = disturb_point(stop)
1867 start = disturb_point(start)
1868 stop = disturb_point(stop)
1869 move(start) curve(start_line,stop_line,stop)
1870 linewidth(drawwidth*(math.random()+0.5))
1871 stroke()
1872 end

```

13 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`.

14 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

15 Literature

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

- Lua_T_EX documentation – the manual and links to presentations and talks: <http://www.luatex.org/documentation.html>
- The Lua manual, for Lua 5.1: <http://www.lua.org/manual/5.1/>
- Programming in Lua, 1st edition, aiming at Lua 5.0, but still (largely) valid for 5.1: <http://www.lua.org/pil/>

16 Thanks

This package would not have been possible without the help of many people who patiently answered my annoying questions on mailing lists and in personal mails. And of course not without the work of the Lua_T_EX team!

Special thanks go to Paul “we could have chickenized the world” Isambert who contributed a lot of ideas, code and bug fixes and made much of the code executable at all. I also thank Philipp Gesang who gave me many advices on the Lua code – which I still didn’t have time to correct ...