CHICKENIZE

v0.3
Arno L. Trautmann A$_T$
arno.trautmann@gmx.de

How to read this document.

This is the documentation of the package `chickenize`. It allows manipulations of any LuaTEX document[1] 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 latet source code is hosted on github: `https://github.com/alt/chickenize`. Feel free to comment or report bugs there, to fork, pull, etc.

---

[1]The code is based on pure LuaTEX features, so don't even try to use it with any other TEX flavour. The package is (partially) tested under plain LuaTEX and (fully) under LuaLATEX. If you tried using it with ConTEXt, 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: åðŭ
randomuclc	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
guttenbergenize	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

---

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

Contents

Part I

User Documentation

1 How It Works

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

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

2 Commands – How You Can Use It

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

2.1 TEX Commands – Document Wide

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

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

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

\boustrophedon Reverts every second line. This immitates archaic greek writings where one line was right-to-left, the next one left-to-right etc.[3] Interestingly, also every glyph was adaptet to the writing direction, so all glyphs are inverted in the right-to-left lines. Actually, there are two versions of this command that differ in their implementation: `\boustrophedon` rotates the whole line, while `\boustrophedonglyphs` changes the writing direction and reverses glyph-wise. The second one takes much more compilation time, but may be more reliable. A Rongorongo[4] similar style boustrophedon is available with `\boustrophedoninverse` or `\rongorongonize`, where subsequent lines are rotated by 180° instead of mirrored.

---

[3] en.wikipedia.org/wiki/Boustrophedon
[4] en.wikipedia.org/wiki/Rongorongo

\countglyphs **\countwords** Counts every printed character (or word, respectively) that appears in anything that is a paragraph. Which is quite everything, in fact, *exept* 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 WOBBBWOBBWOBB BZZZRRRRRRROOOOOOAAAAA ... (inspired by http://www.youtube.com/watch?v=ZFQ5EpO7iHk and http://www.youtube.com/watch?v=nGxpSsbodnw)

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

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

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

\gameoflife Try it.

\hammertime STOP! —— Hammertime!

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

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

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

---

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

\nyanize A synonym for `rainbowcolor`.

\randomerror Just throws a random TeX or LaTeX error at a random time during the compilation. I have quite no idea what this could be used for.

\randomuclc Changes every character of the input into its uppercase or lowercase variant. Well, guess what the "random" means …

\randomfonts Changes the font randomly for every character. If no parameters are given, all fonts that have been loaded are used, especially including math fonts.

\randomcolor Does what its name says.

\rainbowcolor Instead of random colors, this command causes the text color to change gradually according to the colors of a rainbow. Do not mix this with `randomcolor`, as that doesn't make any sense.

\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 occurance 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 `impnattypo` package and based on the same ideas. However, the code in `chickenize` has been written before the author knew `impnattypo`, and the code differs a bit, might even be a bit faster. Well, test it!

\tabularasa Takes every glyph out of the document and replaces it by empty space of the same width. That could be useful if you want to hide some part of a text or similar. The `\text`-version is most likely more useful.

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

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

2.2   How to Deactivate It

Every command has a \un-version that deactivates it's functionality. So once you used \chickenize, it will chickenize the whole document up to \unchickenize. However, the paragraph in which \unchickenize appears, will *not* be chickenized. The same is true for all other manipulations. Take care that you don't \un-anything bevor 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 have[7] 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?

---

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

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

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

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

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

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.

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

3.2 Options for Game of Chicken

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

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

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

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

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

GOCcellcode = **\<string\>** **"scalebox{0.03}{drawchicken}"** The 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\>** **"O"** 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 LuaTEXİt's just to get an idea how things work here. For a deeper understanding of LuaTEX you should consult both the LuaTEX 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 LuaTEX 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 TEXing, 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 calcuation (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 LuaLATEX, 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 LaTeX 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 LuaTeX manual to see what functionality a callback has when executed, what arguments it expects and what return values have to be given.

Everything I have written here is not the complete truth – please consult the LuaTeX manual and the `luatexbase` section in the LaTeX kernel documentation for details!

7 Nodes

Essentially everything that LuaTeX deals with are nodes – letters, spaces, colors, rules etc. In this package, we make heavy use of different types of nodes, so an understanding of the concept is crucial for the functionality.

A node is an object that has different properties, depending on its type which is stored in its `.id` field. For example, a node of type `glyph` has id 27 (up to LuaTeX 0.80, it was 37) has a number `.char` that represents its unicode codepoint, a `.font` entry that determines the font used for this glyph, a `.height`, `.depth` and `.width` etc.

Also, a node typically has a non-empty field `.next` and `.prev`. In a list, these point to the – guess it – next or previous node. Using this, one can walk over a list of nodes step by step and manipulate the list.

A more convenient way to adress each node of a list is the function `node.traverse(head)` which takes as first argument the first node of the list. However, often one wants to adress only a certain type of

---

[9]Since the late 2015 release of LaTeX, the package has not to be loaded anymore since the functionality is absorbed by the kernel. PlainTeX users can load the `ltluatex` 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 LuaTEX manual! Then, you have to remove the `if n.char` condition on the third line of the listing, because glue nodes lack a `.char` field. If everything works, you should have an input consisting of only one long word. Congratulations!

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

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 TEXing or even for good LuaTEXing. It's not. For high quality code check out the code written by Hans Hagen or other professionals. Once you understand the package at hand, you should be ready to go on and improve your knowledge. After that, you might come back and help me improve this package – I'm always happy for any help ☺

---

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

Part III
Implementation

9 TEX file

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

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

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

```
1 \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\BEClerize{
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,"boustrophedo
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,"boustrophe
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,"detectdoublewor
113               luatexbase.add_to_callback("stop_run",printdoublewords,"printdoublewords")
114   }
115 }
116
117 \def\dosomethingfunny{
118     %% should execute one of the "funny" commands, but randomly. So every compilation is completel
  functions. Maybe also on a per-paragraph-basis?
119 }
```

chicken 18

```
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] = "WAAAAAAAAH"
131     chickenstring[8] = "duhduh duhduh duh"
132     chickenstring[9] = "BEEEEEEEEEW"
133     chickenstring[10] = "DDEEEEEEEW"
134     chickenstring[11] = "EEEEEW"
135     chickenstring[12] = "boop"
136     chickenstring[13] = "buhdee"
137     chickenstring[14] = "bee bee"
138     chickenstring[15] = "BZZZRRRRRRROOOOOOOAAAAA"
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\textbacksla
154 \textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbacksla
155 \textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbacksla
  eater}
156 }
157
158 \def\francize{
159   \directlua{luatexbase.add_to_callback("pre_linebreak_filter",francize,"francize")}}
160
161 \def\unfrancize{
162   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter",francize)}}
163
164 \def\gameoflife{
```

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

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

```
167 \def\gameofchicken#1{\directlua{
168 GOCrule_live = {2,3}
169 GOCrule_spawn = {3}
170 GOCrule_antilive = {2,3}
171 GOCrule_antispawn = {3}
172 GOCcellcode = "\\scalebox{0.03}{\\drawchicken}"
173 GOCcellcode = "\\scalebox{0.03}{\\drawcov}"
174 GOCx = 100
175 GOCy = 100
176 GOCiter = 150
177 GOC_console = false
178 GOC_pdf = true
179 GOCsleep = 0
180 GOCdensity = 100
181 #1
182 gameofchicken()
183
184 if (GOCmakegif == true) then
185   luatexbase.add_to_callback("wrapup_run",make_a_gif,"makeagif")
186 end
187 }}
188 \let\gameofchimken\gameofchicken % yeah, that had to be.
189
190 \def\guttenbergenize{ %% 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
213 }
214
215 \def\hammertime{
216   \global\let\n\relax
217   \directlua{hammerfirst = true
218             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,"italianizerand
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{lutaexbase.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,"letterspaceadjus
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}}
```

chicken 22

```
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_col
341 \def\unscorpionize{
342   \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","scorpionize_color")}}
343
344 \def\spankmonkey{    %% to be implemented
345   \directlua{}}
346 \def\unspankmonkey{
```

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

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

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

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

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

```
434 \def\chickenizesetup#1{\directlua{#1}}
```

chicken 25

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.

```
435 \long\def\luadraw#1#2{%
436   \vbox to #1bp{%
437     \vfil
438     \latelua{pdf_print("q") #2 pdf_print("Q")}%
439   }%
440 }
441 \long\def\drawchicken{
442   \luadraw{90}{
443     chickenhead     = {200,50} % chicken head center
444     chickenhead_rad = 20
445
446     neckstart = {215,35} % neck
447     neckstop  = {230,10} %
448
449     chickenbody     = {260,-10}
450     chickenbody_rad = 40
451     chickenleg = {
452       {{260,-50},{250,-70},{235,-70}},
453       {{270,-50},{260,-75},{245,-75}}
454     }
455
456     beak_top = {185,55}
457     beak_front = {165,45}
458     beak_bottom = {185,35}
459
460     wing_front = {260,-10}
461     wing_bottom = {280,-40}
462     wing_back = {275,-15}
463
464     sloppycircle(chickenhead,chickenhead_rad) sloppyline(neckstart,neckstop)
465     sloppycircle(chickenbody,chickenbody_rad)
466     sloppyline(chickenleg[1][1],chickenleg[1][2]) sloppyline(chickenleg[1][2],chickenleg[1][3])
467     sloppyline(chickenleg[2][1],chickenleg[2][2]) sloppyline(chickenleg[2][2],chickenleg[2][3])
468     sloppyline(beak_front,beak_top) sloppyline(beak_front,beak_bottom)
469     sloppyline(wing_front,wing_bottom) sloppyline(wing_back,wing_bottom)
470   }
471 }
```

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.

chicken 26

```
472 \long\def\drawcov{
473   \luadraw{90}{
474     covbody = {200,50}
475     covbody_rad = 50
476
477     covcrown_rad = 5
478     crownno = 13
479     for i=1,crownno do
480       crownpos = {covbody[1]+1.4*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+1.4*covbody_
481       crownconnect = {covbody[1]+covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+covbody_rad
482       sloppycircle(crownpos,covcrown_rad)
483       sloppyline(crownpos,crownconnect)
484     end
485
486     covcrown_rad = 6
487     crownno = 8
488     for i=1,crownno do
489       crownpos = {covbody[1]+0.8*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+0.8*covbody_
490       crownconnect = {covbody[1]+0.5*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+0.5*covb
491       sloppycircle(crownpos,covcrown_rad)
492       sloppyline(crownpos,crownconnect)
493     end
494
495     covcrown_rad = 8
496     sloppycircle(covbody,covcrown_rad)
497     sloppycircle(covbody,covbody_rad)
498     sloppyline(covbody,covbody)
499   }
500 }
```

9.4 drawhorse

Well … guess what this does.

```
501 \long\def\drawhorse{
502   \luadraw{90}{
503     horsebod = {100,-40}
504     sloppyellipsis(horsebod,50,20)
505     horsehead = {20,0}
506     sloppyellipsis(horsehead,25,15)
507     sloppyline({35,-10},{50,-40})
508     sloppyline({45,5},{80,-25})
509     sloppyline({60,-50},{60,-90})
510     sloppyline({70,-50},{70,-90})
511     sloppyline({130,-50},{130,-90})
512     sloppyline({140,-50},{140,-90})
513     sloppyline({150,-40},{160,-60})
```

```
514    sloppyline({150,-38},{160,-58})
515    sloppyline({150,-42},{160,-62})
516    sloppyline({-5,-10},{10,-5})
517    sloppyellipsis({30,5},5,2)  %% it's an eye, aye?
518    sloppyline({27,15},{34,25})
519    sloppyline({34,25},{37,13})
520  }
521 }
```

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

```
522 \long\def\drawfathorse{
523  \luadraw{90}{
524    horsebod = {100,-40}
525    sloppyellipsis(horsebod,50,40)
526    horsehead = {20,0}
527    sloppyellipsis(horsehead,25,15)
528    sloppyline({35,-10},{50,-40})
529    sloppyline({45,5},{70,-15})
530    sloppyline({60,-70},{60,-90})
531    sloppyline({70,-70},{70,-90})
532    sloppyline({130,-70},{130,-90})
533    sloppyline({140,-70},{140,-90})
534    sloppyline({150,-40},{160,-60})
535    sloppyline({150,-38},{160,-58})
536    sloppyline({150,-42},{160,-62})
537    sloppyline({-5,-10},{10,-5})
538    sloppyellipsis({30,5},5,2)  %% it's an eye, aye?
539    sloppyline({27,15},{34,25})
540    sloppyline({34,25},{37,13})
541  }
542 }
543 % intentioally not documented:
544 \long\def\drawunicorn{
545 \color{pink!90!black}
546  \drawhorse
547  \luadraw{0}{
548    sloppyline({15,20},{15,50})
549    sloppyline({15,50},{25,20})
550  }
551 }
552 \long\def\drawfatunicorn{
553 \color{pink!90!black}
554  \drawfathorse
555  \luadraw{0}{
556    sloppyline({15,20},{15,50})
557    sloppyline({15,50},{25,20})
558  }
```

```
559 }
```

10   LᴬTᴇX package

I have decided to keep the LᴬTᴇX-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.

```
560 \ProvidesPackage{chickenize}%
561   [2021/01/03 v0.3 chickenize package]
562 \input{chickenize}
```

10.1   Free Compliments

```
563 %
```

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! :)

```
564 \iffalse
565   \DeclareDocumentCommand\includegraphics{O{}m}{
566     \fbox{Chicken}  %% actually, I'd love to draw an MP graph showing a chicken …
567   }
568 %%%% specials: the balmerpeak. A tribute to http://xkcd.com/323/.
569 %% So far, you have to load pgfplots yourself.
570 %% As it is a mighty package, I don't want the user to force loading it.
571 \NewDocumentCommand\balmerpeak{G{}O{-4cm}}{
572 %% to be done using Lua drawing.
573 }
574 \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.

```
575
576 local nodeid   = node.id
577 local nodecopy = node.copy
578 local nodenew  = node.new
579 local nodetail = node.tail
```

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

```
599 color_push = nodenew(WHAT,COL)
600 color_pop = nodenew(WHAT,COL)
601 color_push.stack = 0
602 color_pop.stack = 0
603 color_push.command = 1
604 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.

```
605 chicken_pagenumbers = true
606
607 chickenstring = {}
608 chickenstring[1] = "chicken" -- chickenstring is a table, please remeber this!
609
610 chickenizefraction = 0.5 -- set this to a small value to fool somebody,
611 -- or to see if your text has been read carefully. This is also a great way to lay easter eggs fo
612 chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing you
613
614 local match = unicode.utf8.match
615 chickenize_ignore_word = false
```

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

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

```
617     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
618       i.next = i.next.next
619     end
620
621     chicken = {}  -- constructing the node list.
622
623 -- Should this be done only once? No, otherwise we lose the freedom to change the string in-
   document.
624 -- But it could be done only once each paragraph as in-paragraph changes are not possible!
625
626     chickenstring_tmp = chickenstring[math.random(1,#chickenstring)]
627     chicken[0] = nodenew(GLYPH,1)  -- only a dummy for the loop
628     for i = 1,string.len(chickenstring_tmp) do
629       chicken[i] = nodenew(GLYPH,1)
630       chicken[i].font = font.current()
631       chicken[i-1].next = chicken[i]
632     end
633
634     j = 1
635     for s in string.utfvalues(chickenstring_tmp) do
636       local char = unicode.utf8.char(s)
637       chicken[j].char = s
638       if match(char,"%s") then
639         chicken[j] = nodenew(GLUE)
640         chicken[j].width = space
641         chicken[j].shrink = shrink
642         chicken[j].stretch = stretch
643       end
644       j = j+1
645     end
646
647     nodeslide(chicken[1])
648     lang.hyphenate(chicken[1])
649     chicken[1] = node.kerning(chicken[1])    -- FIXME: does not work
650     chicken[1] = node.ligaturing(chicken[1]) -- dito
651
652     nodeinsertbefore(head,i,chicken[1])
653     chicken[1].next = chicken[2] -- seems to be necessary … to be fixed
654     chicken[string.len(chickenstring_tmp)].next = i.next
655
656     -- shift lowercase latin letter to uppercase if the original input was an uppercase
657     if (chickenize_capital and (chicken[1].char > 96 and chicken[1].char < 123)) then
658       chicken[1].char = chicken[1].char - 32
659     end
660
```

```
661    return head
662 end
663
664 chickenize = function(head)
665   for i in nodetraverseid(GLYPH,head) do  --find start of a word
666     -- Random determination of the chickenization of the next word:
667     if math.random() > chickenizefraction then
668       chickenize_ignore_word = true
669     elseif chickencount then
670       chicken_substitutions = chicken_substitutions + 1
671     end
672
673     if (chickenize_ignore_word == false) then  -- normal case: at the beginning of a word, we jum
674       if (i.char > 64 and i.char < 91) then chickenize_capital = true else chickenize_capital = fa
675       head = chickenize_real_stuff(i,head)
676     end
677
678 -- At the end of the word, the ignoring is reset. New chance for everyone.
679     if not((i.next.id == GLYPH) or (i.next.id == DISC) or (i.next.id == PUNCT) or (i.next.id == K
680       chickenize_ignore_word = false
681     end
682   end
683   return head
684 end
685
```

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)

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

11.2 boustrophedon

There are two implementations of the boustrophedon: One reverses every line as a whole, the other one changes the writing direction and reverses glyphs one by one. The latter one might be more reliable, but takes considerably more time.

Linewise rotation:

```
702 boustrophedon = function(head)
703   rot = node.new(WHAT,PDF_LITERAL)
704   rot2 = node.new(WHAT,PDF_LITERAL)
705   odd = true
706     for line in node.traverse_id(0,head) do
707       if odd == false then
708         w = line.width/65536*0.99625 -- empirical correction factor (?)
709         rot.data  = "-1 0 0 1 "..w.." 0 cm"
710         rot2.data = "-1 0 0 1 "..-w.." 0 cm"
711         line.head = node.insert_before(line.head,line.head,nodecopy(rot))
712         nodeinsertafter(line.head,nodetail(line.head),nodecopy(rot2))
713         odd = true
714       else
715         odd = false
716       end
717     end
718   return head
719 end
```

Glyphwise rotation:

```
720 boustrophedon_glyphs = function(head)
721   odd = false
722   rot = nodenew(WHAT,PDF_LITERAL)
723   rot2 = nodenew(WHAT,PDF_LITERAL)
724   for line in nodetraverseid(0,head) do
725     if odd==true then
726       line.dir = "TRT"
727       for g in nodetraverseid(GLYPH,line.head) do
728         w = -g.width/65536*0.99625
729         rot.data = "-1 0 0 1 " .. w .." 0 cm"
730         rot2.data = "-1 0 0 1 " .. -w .." 0 cm"
731         line.head = node.insert_before(line.head,g,nodecopy(rot))
732         nodeinsertafter(line.head,g,nodecopy(rot2))
733       end
734       odd = false
735     else
736       line.dir = "TLT"
737       odd = true
738     end
739   end
740   return head
```

```
741 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.

```
742 boustrophedon_inverse = function(head)
743   rot = node.new(WHAT,PDF_LITERAL)
744   rot2 = node.new(WHAT,PDF_LITERAL)
745   odd = true
746     for line in node.traverse_id(0,head) do
747       if odd == false then
748 texio.write_nl(line.height)
749         w = line.width/65536*0.99625 -- empirical correction factor (?)
750         h = line.height/65536*0.99625
751         rot.data  = "-1 0 0 -1 "..w.." "..h.." cm"
752         rot2.data = "-1 0 0 -1 "..-w.." "..0.5*h.." cm"
753         line.head = node.insert_before(line.head,line.head,node.copy(rot))
754         node.insert_after(line.head,node.tail(line.head),node.copy(rot2))
755         odd = true
756       else
757         odd = false
758       end
759     end
760   return head
761 end
```

11.3 bubblesort

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

```
762 function bubblesort(head)
763   for line in nodetraverseid(0,head) do
764     for glyph in nodetraverseid(GLYPH,line.head) do
765
766     end
767   end
768   return head
769 end
```

11.4 countglyphs

Counts the glyphs in your document. Where "glyph" means every printed character in everything that is a paragraph – formulas do *not* work! Captions of floats etc. also will *not* work. However, hyphenations *do* work and the hyphen sign *is counted*! And that is the sole reason for this function – every simple script could read the letters in a doucment, 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!

```
770 countglyphs = function(head)
771   for line in nodetraverseid(0,head) do
772     for glyph in nodetraverseid(GLYPH,line.head) do
773       glyphnumber = glyphnumber + 1
774       if (glyph.next.next) then
775         if (glyph.next.id == 10) and (glyph.next.next.id == GLYPH) then
776           spacenumber = spacenumber + 1
777         end
778         counted_glyphs_by_code[glyph.char] = counted_glyphs_by_code[glyph.char] + 1
779       end
780     end
781   end
782   return head
783 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? …

```
784 printglyphnumber = function()
785   texiowrite_nl("\nNumber of glyphs by character code (only up to 127):")
786   for i = 1,127 do --%% FIXME: should allow for more characters, but cannot be printed to console
787     texiowrite_nl(string.char(i)..": "..counted_glyphs_by_code[i])
788   end
789
790   texiowrite_nl("\nTotal number of glyphs in this document: "..glyphnumber)
791   texiowrite_nl("Number of spaces in this document: "..spacenumber)
792   texiowrite_nl("Glyphs plus spaces: "..glyphnumber+spacenumber.."\n")
793 end
```

11.5   countwords

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

```
794 countwords = function(head)
795   for glyph in nodetraverseid(GLYPH,head) do
796     if (glyph.next.id == GLUE) then
797       wordnumber = wordnumber + 1
798     end
799   end
```

```
800   wordnumber = wordnumber + 1 -- add 1 for the last word in a paragraph which is not found otherwi
801   return head
802 end
```

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

```
803 printwordnumber = function()
804   texiowrite_nl("\nNumber of words in this document: "..wordnumber)
805 end
```

11.6   detectdoublewords

```
806 %% FIXME: Does this work? …
807 detectdoublewords = function(head)
808   prevlastword  = {}  -- array of numbers representing the glyphs
809   prevfirstword = {}
810   newlastword   = {}
811   newfirstword  = {}
812   for line in nodetraverseid(0,head) do
813     for g in nodetraverseid(GLYPH,line.head) do
814 texio.write_nl("next glyph",#newfirstword+1)
815       newfirstword[#newfirstword+1] = g.char
816       if (g.next.id == 10) then break end
817     end
818 texio.write_nl("nfw:"..#newfirstword)
819   end
820 end
821
822 printdoublewords = function()
823   texio.write_nl("finished")
824 end
```

11.7   francize

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

```
825 francize = function(head)
826   for n in nodetraverseid(GLYPH,head) do
827     if ((n.char > 47) and (n.char < 58)) then
828       n.char = math.random(48,57)
829     end
830   end
831   return head
832 end
```

11.8   gamofchicken

The `gameofchicken` is an implementation of the Game of Life by Conway. The standard cell here is a chicken, while there are also anticells. For both you can adapt the 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 LuaTEX 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.

```
833 function gameofchicken()
834   GOC_lifetab = {}
835   GOC_spawntab = {}
836   GOC_antilifetab = {}
837   GOC_antispawntab = {}
838   -- translate the rules into an easily-manageable table
839   for i=1,#GOCrule_live do; GOC_lifetab[GOCrule_live[i]] = true end
840   for i=1,#GOCrule_spawn do; GOC_spawntab[GOCrule_spawn[i]] = true end
841   for i=1,#GOCrule_antilive do; GOC_antilifetab[GOCrule_antilive[i]] = true end
842   for i=1,#GOCrule_antispawn do; GOC_antispawntab[GOCrule_antispawn[i]] = true end
```

Initialize the arrays for cells and anticells with zeros.

```
843 -- initialize the arrays
844 local life = {}
845 local antilife = {}
846 local newlife = {}
847 local newantilife = {}
848 for i = 0, GOCx do life[i] = {}; newlife[i] = {} for j = 0, GOCy do life[i][j] = 0 end end
849 for i = 0, GOCx do antilife[i] = {}; newantilife[i] = {} for j = 0, GOCy do antilife[i][j] = 0 end
```

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

```
850 function applyruleslife(neighbors, lifeij, antineighbors, antilifeij)
851   if GOC_spawntab[neighbors] then myret = 1 else -- new cell
852   if GOC_lifetab[neighbors] and (lifeij == 1) then myret = 1 else myret =  0 end end
853   if antineighbors > 1 then myret =  0 end
854   return myret
855 end
856 function applyrulesantilife(neighbors, lifeij, antineighbors, antilifeij)
857   if (antineighbors == 3) then myret = 1 else -- new cell or keep cell
858   if (((antineighbors > 1) and (antineighbors < 4)) and (lifeij == 1)) then myret = 1 else myret =
859   if neighbors > 1 then myret =  0 end
860   return myret
861 end
```

Preparing the initial state with a default pattern:

```
862 -- prepare some special patterns as starter
863 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:

```
864   print("start");
865   for i = 1,GOCx do
866     for j = 1,GOCy do
867       if (life[i][j]==1) then texio.write("X") else if (antilife[i][j]==1) then texio.write("O")
868     end
```

```
869     texio.write_nl(" ");
870   end
871   os.sleep(GOCsleep)
872
873   for i = 0, GOCx do
874     for j = 0, GOCy do
875         newlife[i][j] = 0 -- Fill the values from the start settings here
876         newantilife[i][j] = 0 -- Fill the values from the start settings here
877     end
878   end
879
880   for k = 1,GOCiter do -- iterate over the cycles
881     texio.write_nl(k);
882     for i = 1, GOCx-1 do -- iterate over lines
883       for j = 1, GOCy-1 do -- iterate over columns -- prevent edge effects
884         local neighbors = (life[i-1][j-1] + life[i-1][j] + life[i-1][j+1] + life[i][j-
  1] + life[i][j+1] +  life[i+1][j-1] + life[i+1][j] + life[i+1][j+1])
885         local antineighbors = (antilife[i-1][j-1] + antilife[i-1][j] + antilife[i-
  1][j+1] + antilife[i][j-1] + antilife[i][j+1] +  antilife[i+1][j-1] + antilife[i+1][j] + antilife
886
887         newlife[i][j] = applyruleslife(neighbors, life[i][j],antineighbors, antilife[i][j])
888         newantilife[i][j] = applyrulesantilife(neighbors,life[i][j], antineighbors,antilife[i][j])
889       end
890     end
891
892     for i = 1, GOCx do
893       for j = 1, GOCy do
894         life[i][j] = newlife[i][j] -- copy the values
895         antilife[i][j] = newantilife[i][j] -- copy the values
896       end
897     end
898
899     for i = 1,GOCx do
900       for j = 1,GOCy do
901         if GOC_console then
902           if (life[i][j]==1) then texio.write("X") else if (antilife[i][j]==1) then texio.write("
903         end
904         if GOC_pdf then
905           if (life[i][j]==1) then tex.print("\\placeat("..(i/10)..","..(j/10)..")}{"..GOCcellcode.
906           if (antilife[i][j]==1) then tex.print("\\placeat("..(i/10)..","..(j/10)..")}{"..GOCantice
907         end
908       end
909     end
910     tex.print(".\\newpage")
911     os.sleep(GOCsleep)
912   end
```

```
913 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.

```
914 function make_a_gif()
915   os.execute("convert -verbose -dispose previous -background white -alpha remove -
      alpha off -density "..GOCdensity.." " ..tex.jobname ..".pdf " ..tex.jobname..".gif")
916   os.execute("gwenview "..tex.jobname..".gif")
917 end
```

11.9   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.

11.9.1   guttenbergenize – preliminaries

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

```
918 local quotestrings = {
919   [171] = true,  [172] = true,
920   [8216] = true, [8217] = true, [8218] = true,
921   [8219] = true, [8220] = true, [8221] = true,
922   [8222] = true, [8223] = true,
923   [8248] = true, [8249] = true, [8250] = true,
924 }
```

11.9.2   guttenbergenize – the function

```
925 guttenbergenize_rq = function(head)
926   for n in nodetraverseid(GLYPH,head) do
927     local i = n.char
928     if quotestrings[i] then
929       noderemove(head,n)
930     end
931   end
932   return head
933 end
```

---

[11]Thanks to Jasper for bringing me to this idea!

11.10   hammertime

This is a completely useless function. It just prints STOP! – HAMMERTIME at the beginnig of the first paragraph after \hammertime, and "U can't touch this" for every following one. As the function writes to the terminal, you have to be sure that your terminal is line-buffered and not block-buffered. Compare the explanation by Taco on the LuaTEX mailing list.[12]

```
934 hammertimedelay = 1.2
935 local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
936 hammertime = function(head)
937   if hammerfirst then
938     texiowrite_nl(htime_separator)
939     texiowrite_nl("============STOP!============\n")
940     texiowrite_nl(htime_separator .. "\n\n\n")
941     os.sleep (hammertimedelay*1.5)
942     texiowrite_nl(htime_separator .. "\n")
943     texiowrite_nl("=========HAMMERTIME=========\n")
944     texiowrite_nl(htime_separator .. "\n\n")
945     os.sleep (hammertimedelay)
946     hammerfirst = false
947   else
948     os.sleep (hammertimedelay)
949     texiowrite_nl(htime_separator)
950     texiowrite_nl("======U can't touch this!=====\n")
951     texiowrite_nl(htime_separator .. "\n\n")
952     os.sleep (hammertimedelay*0.5)
953   end
954   return head
955 end
```

11.11   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.

```
956 italianizefraction = 0.5 --%% gives the amount of italianization
957 mynode = nodenew(GLYPH) -- prepare a dummy glyph
958
959 italianize = function(head)
960   -- skip "h/H" randomly
961   for n in node.traverse_id(GLYPH,head) do -- go through all glyphs
962       if n.prev.id ~= GLYPH then -- check if it's a word start
963       if ((n.char == 72) or (n.char == 104)) and (tex.normal_rand() < italianizefraction) then --
964         n.prev.next = n.next
965       end
```

---

[12]http://tug.org/pipermail/luatex/2011-November/003355.html

```
966      end
967    end
968
969    -- add h or H in front of vowels
970    for n in nodetraverseid(GLYPH,head) do
971      if math.random() < italianizefraction then
972      x = n.char
973      if x == 97 or x == 101 or x == 105 or x == 111 or x == 117 or
974         x == 65 or x ==  69 or x ==  73 or x == 79 or x == 85 then
975        if (n.prev.id == GLUE) then
976          mynode.font = n.font
977          if x > 90 then  -- lower case
978            mynode.char = 104
979          else
980            mynode.char = 72 -- upper case - convert into lower case
981            n.char = x + 32
982          end
983            node.insert_before(head,n,node.copy(mynode))
984          end
985        end
986      end
987    end
988
989    -- add e after words, but only after consonants
990    for n in node.traverse_id(GLUE,head) do
991      if n.prev.id == GLYPH then
992      x = n.prev.char
993      -- skip vowels and randomize
994      if not(x == 97 or x == 101 or x == 105 or x == 111 or x == 117 or x == 44 or x == 46) and matl
995          mynode.char = 101            -- it's always a lower case e, no?
996          mynode.font = n.prev.font -- adapt the current font
997          node.insert_before(head,n,node.copy(mynode)) -- insert the e in the node list
998        end
999      end
1000   end
1001
1002   return head
1003 end
```

11.12   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.

```
1004 italianizerandwords = function(head)
1005 words = {}
```

```
1006 wordnumber = 0
1007 -- head.next.next is the very first word. However, let's try to get the first word after the first
1008   for n in nodetraverseid(GLUE,head) do -- let's try to count words by their separators
1009     wordnumber = wordnumber + 1
1010     if n.next then
1011       words[wordnumber] = {}
1012       words[wordnumber][1] = node.copy(n.next)
1013
1014       glyphnumber = 1
1015       myglyph = n.next
1016       while myglyph.next do
1017         node.tail(words[wordnumber][1]).next = node.copy(myglyph.next)
1018         myglyph = myglyph.next
1019       end
1020     end
1021   print(#words)
1022   if #words > 0 then
1023   print("lengs is: ")
1024   print(#words[#words])
1025   end
1026   end
1027 --myinsertnode = head.next.next -- first letter
1028 --node.tail(words[1][1]).next = myinsertnode.next
1029 --myinsertnode.next = words[1][1]
1030
1031   return head
1032 end
1033
1034 italianize_old = function(head)
1035   local wordlist = {} -- here we will store the number of words of the sentence.
1036   local words = {} -- here we will store the words of the sentence.
1037   local wordnumber = 0
1038   -- let's first count all words in one sentence, howboutdat?
1039   wordlist[wordnumber] = 1 -- let's save the word *length* in here …
1040
1041
1042   for n in nodetraverseid(GLYPH,head) do
1043     if (n.next.id == GLUE) then -- this is a space
1044       wordnumber = wordnumber + 1
1045       wordlist[wordnumber] = 1
1046       words[wordnumber] = n.next.next
1047     end
1048     if (n.next.id == GLYPH) then  -- it's a glyph
1049     if (n.next.char == 46) then -- this is a full stop.
1050       wordnumber = wordnumber + 1
1051       texio.write_nl("this sentence had "..wordnumber.."words.")
```

```
1052        for i=0,wordnumber-1 do
1053        texio.write_nl("word "..i.." had " .. wordlist[i] .. "glyphs")
1054        end
1055        texio.write_nl(" ")
1056        wordnumber = -1 -- to compensate the fact that the next node will be a space, this would cou
1057     else
1058
1059        wordlist[wordnumber] = wordlist[wordnumber] + 1 -- the current word got 1 glyph longer
1060        end
1061     end
1062   end
1063   return head
1064 end
```

11.13   itsame

The (very first, very basic, very stupid) code to draw a small mario. You need to input luadraw.tex or do
luadraw.lua for the rectangle function.

```
1065 itsame = function()
1066 local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
1067 color = "1 .6 0"
1068 for i = 6,9 do mr(i,3) end
1069 for i = 3,11 do mr(i,4) end
1070 for i = 3,12 do mr(i,5) end
1071 for i = 4,8 do mr(i,6) end
1072 for i = 4,10 do mr(i,7) end
1073 for i = 1,12 do mr(i,11) end
1074 for i = 1,12 do mr(i,12) end
1075 for i = 1,12 do mr(i,13) end
1076
1077 color = ".3 .5 .2"
1078 for i = 3,5 do mr(i,3) end mr(8,3)
1079 mr(2,4) mr(4,4) mr(8,4)
1080 mr(2,5) mr(4,5) mr(5,5) mr(9,5)
1081 mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
1082 for i = 3,8 do mr(i,8) end
1083 for i = 2,11 do mr(i,9) end
1084 for i = 1,12 do mr(i,10) end
1085 mr(3,11) mr(10,11)
1086 for i = 2,4 do mr(i,15) end for i = 9,11 do mr(i,15) end
1087 for i = 1,4 do mr(i,16) end for i = 9,12 do mr(i,16) end
1088
1089 color = "1 0 0"
1090 for i = 4,9 do mr(i,1) end
1091 for i = 3,12 do mr(i,2) end
1092 for i = 8,10 do mr(5,i) end
```

```
1093 for i = 5,8 do mr(i,10) end
1094 mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
1095 for i = 4,9 do mr(i,12) end
1096 for i = 3,10 do mr(i,13) end
1097 for i = 3,5 do mr(i,14) end
1098 for i = 7,10 do mr(i,14) end
1099 end
```

11.14   kernmanipulate

This function either eliminates all the kerning, inverts the sign of the kerning or changes it to a user-given value.

If the boolean `chickeninvertkerning` is true, the kerning amount is negative, if it is false, the kerning will be set to th e value of `chickenkernvalue`. A large value (> 100 000) can be used to show explicitely where kerns are inserted. Good for educational use.

```
1100 chickenkernamount = 0
1101 chickeninvertkerning = false
1102
1103 function kernmanipulate (head)
1104   if chickeninvertkerning then -- invert the kerning
1105     for n in nodetraverseid(11,head) do
1106       n.kern = -n.kern
1107     end
1108   else             -- if not, set it to the given value
1109     for n in nodetraverseid(11,head) do
1110       n.kern = chickenkernamount
1111     end
1112   end
1113   return head
1114 end
```

11.15   leetspeak

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

```
1115 leetspeak_onlytext = false
1116 leettable = {
1117   [101] = 51, -- E
1118   [105] = 49, -- I
1119   [108] = 49, -- L
1120   [111] = 48, -- O
1121   [115] = 53, -- S
1122   [116] = 55, -- T
1123
1124   [101-32] = 51, -- e
1125   [105-32] = 49, -- i
```

```
1126   [108-32] = 49, -- l
1127   [111-32] = 48, -- o
1128   [115-32] = 53, -- s
1129   [116-32] = 55, -- t
1130 }
```

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

```
1131 leet = function(head)
1132   for line in nodetraverseid(Hhead,head) do
1133     for i in nodetraverseid(GLYPH,line.head) do
1134       if not leetspeak_onlytext or
1135          node.has_attribute(i,luatexbase.attributes.leetattr)
1136       then
1137         if leettable[i.char] then
1138           i.char = leettable[i.char]
1139         end
1140       end
1141     end
1142   end
1143   return head
1144 end
```

11.16   leftsideright

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

```
1145 leftsideright = function(head)
1146   local factor = 65536/0.99626
1147   for n in nodetraverseid(GLYPH,head) do
1148     if (leftsiderightarray[n.char]) then
1149       shift = nodenew(WHAT,PDF_LITERAL)
1150       shift2 = nodenew(WHAT,PDF_LITERAL)
1151       shift.data = "q -1 0 0 1 " .. n.width/factor .." 0 cm"
1152       shift2.data = "Q 1 0 0 1 " .. n.width/factor .." 0 cm"
1153       nodeinsertbefore(head,n,shift)
1154       nodeinsertafter(head,n,shift2)
1155     end
1156   end
1157   return head
1158 end
```

11.17   letterspaceadjust

Yet another piece of code by Paul. This is primarily intended for very narrow columns, but may also increase the overall quality of typesetting. Basically, it does nothing else than adding expandable space *between* letters. This way, the amount of stretching between words can be reduced which will, hopefully, result in the greyness to be more equally distributed over the page.

Why the synonym `stealsheep`? Because of a comment of Paul on the `texhax` mailing list:

11.17.1   setup of variables

```
1159 local letterspace_glue   = nodenew(GLUE)
1160 local letterspace_pen    = nodenew(PENALTY)
1161
1162 letterspace_glue.width   = tex.sp"0pt"
1163 letterspace_glue.stretch = tex.sp"0.5pt"
1164 letterspace_pen.penalty  = 10000
```

11.17.2   function implementation

```
1165 letterspaceadjust = function(head)
1166   for glyph in nodetraverseid(GLYPH, head) do
1167     if glyph.prev and (glyph.prev.id == GLYPH or glyph.prev.id == DISC or glyph.prev.id == KERN) t
1168       local g = nodecopy(letterspace_glue)
1169       nodeinsertbefore(head, glyph, g)
1170       nodeinsertbefore(head, g, nodecopy(letterspace_pen))
1171     end
1172   end
1173   return head
1174 end
```

11.17.3   textletterspaceadjust

The `\text...`-version of `letterspaceadjust`. Just works, without the need to call `\letterspaceadjust` globally or anything else. Just put the `\textletterspaceadjust` around the part of text you want the function to work on. Might have problems with surrounding spacing, take care!

```
1175 textletterspaceadjust = function(head)
1176   for glyph in nodetraverseid(GLYPH, head) do
1177     if node.has_attribute(glyph,luatexbase.attributes.letterspaceadjustattr) then
1178       if glyph.prev and (glyph.prev.id == node.id"glyph" or glyph.prev.id == node.id"disc" or glyp
1179         local g = node.copy(letterspace_glue)
1180         nodeinsertbefore(head, glyph, g)
1181         nodeinsertbefore(head, g, nodecopy(letterspace_pen))
1182       end
1183     end
1184   end
1185   luatexbase.remove_from_callback("pre_linebreak_filter","textletterspaceadjust")
1186   return head
1187 end
```

11.18   matrixize

Substitutes every glyph by a representation of its ASCII value. Migth 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.

```
1188 matrixize = function(head)
1189   x = {}
1190   s = nodenew(DISC)
1191   for n in nodetraverseid(GLYPH,head) do
1192     j = n.char
1193     for m = 0,7 do -- stay ASCII for now
1194       x[7-m] = nodecopy(n) -- to get the same font etc.
1195
1196       if (j / (2^(7-m)) < 1) then
1197         x[7-m].char = 48
1198       else
1199         x[7-m].char = 49
1200         j = j-(2^(7-m))
1201       end
1202       nodeinsertbefore(head,n,x[7-m])
1203       nodeinsertafter(head,x[7-m],nodecopy(s))
1204     end
1205     noderemove(head,n)
1206   end
1207   return head
1208 end
```

11.19   medievalumlaut

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

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

```
1209 medievalumlaut = function(head)
1210   local factor = 65536/0.99626
1211   local org_e_node = nodenew(GLYPH)
1212   org_e_node.char = 101
1213   for line in nodetraverseid(0,head) do
1214     for n in nodetraverseid(GLYPH,line.head) do
1215       if (n.char == 228 or n.char == 246 or n.char == 252) then
1216         e_node = nodecopy(org_e_node)
1217         e_node.font = n.font
1218         shift = nodenew(WHAT,PDF_LITERAL)
1219         shift2 = nodenew(WHAT,PDF_LITERAL)
1220         shift2.data = "Q 1 0 0 1 " .. e_node.width/factor .." 0 cm"
1221         nodeinsertafter(head,n,e_node)
1222
1223         nodeinsertbefore(head,e_node,shift)
```

```
1224        nodeinsertafter(head,e_node,shift2)
1225
1226        x_node = nodenew(KERN)
1227        x_node.kern = -e_node.width
1228        nodeinsertafter(head,shift2,x_node)
1229      end
1230
1231      if (n.char == 228) then -- ä
1232        shift.data = "q 0.5 0 0 0.5 " ..
1233          -n.width/factor*0.85 .." " .. n.height/factor*0.75 .. " cm"
1234        n.char = 97
1235      end
1236      if (n.char == 246) then -- ö
1237        shift.data = "q 0.5 0 0 0.5 " ..
1238          -n.width/factor*0.75 .." " .. n.height/factor*0.75 .. " cm"
1239        n.char = 111
1240      end
1241      if (n.char == 252) then -- ü
1242        shift.data = "q 0.5 0 0 0.5 " ..
1243          -n.width/factor*0.75 .." " .. n.height/factor*0.75 .. " cm"
1244        n.char = 117
1245      end
1246    end
1247  end
1248  return head
1249 end
```

11.20 pancakenize

```
1250 local separator     = string.rep("=", 28)
1251 local texiowrite_nl = texio.write_nl
1252 pancaketext = function()
1253   texiowrite_nl("Output written on "..tex.jobname..".pdf ("..status.total_pages.." chicken,".." eg
1254   texiowrite_nl(" ")
1255   texiowrite_nl(separator)
1256   texiowrite_nl("Soo ... you decided to use \\pancakenize.")
1257   texiowrite_nl("That means you owe me a pancake!")
1258   texiowrite_nl(" ")
1259   texiowrite_nl("(This goes by document, not compilation.)")
1260   texiowrite_nl(separator.."\n\n")
1261   texiowrite_nl("Looking forward for my pancake! :)")
1262   texiowrite_nl("\n\n")
1263 end
```

11.21 randomerror

Not yet implemented, sorry.

11.22   randomfonts

Traverses the output and substitutes fonts randomly. A check is done so that the font number is existing.
One day, the fonts should be easily given explicitly in terms of \bf etc.

```
1264 randomfontslower = 1
1265 randomfontsupper = 0
1266 %
1267 randomfonts = function(head)
1268   local rfub
1269   if randomfontsupper > 0 then  -- fixme: this should be done only once, no? Or at every paragraph
1270     rfub = randomfontsupper  -- user-specified value
1271   else
1272     rfub = font.max()        -- or just take all fonts
1273   end
1274   for line in nodetraverseid(Hhead,head) do
1275     for i in nodetraverseid(GLYPH,line.head) do
1276       if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattr) then
1277         i.font = math.random(randomfontslower,rfub)
1278       end
1279     end
1280   end
1281   return head
1282 end
```

11.23   randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```
1283 uclcratio = 0.5 -- ratio between uppercase and lower case
1284 randomuclc = function(head)
1285   for i in nodetraverseid(GLYPH,head) do
1286     if not(randomuclc_onlytext) or node.has_attribute(i,luatexbase.attributes.randuclcattr) then
1287       if math.random() < uclcratio then
1288         i.char = tex.uccode[i.char]
1289       else
1290         i.char = tex.lccode[i.char]
1291       end
1292     end
1293   end
1294   return head
1295 end
```

11.24   randomchars

```
1296 randomchars = function(head)
1297   for line in nodetraverseid(Hhead,head) do
1298     for i in nodetraverseid(GLYPH,line.head) do
```

```
1299        i.char = math.floor(math.random()*512)
1300     end
1301  end
1302  return head
1303 end
```

11.25   randomcolor and rainbowcolor

11.25.1   randomcolor – preliminaries

Setup of the boolean for grey/color or rainbowcolor, and boundaries for the colors. RGB space is fully used, but greyscale is only used in a visible range, i. e. to 90% instead of 100% white.

```
1304 randomcolor_grey = false
1305 randomcolor_onlytext = false --switch between local and global colorization
1306 rainbowcolor = false
1307
1308 grey_lower = 0
1309 grey_upper = 900
1310
1311 Rgb_lower = 1
1312 rGb_lower = 1
1313 rgB_lower = 1
1314 Rgb_upper = 254
1315 rGb_upper = 254
1316 rgB_upper = 254
```

Variables for the rainbow. 1/rainbow_step*5 is the number of letters used for one cycle, the color changes from red to yellow to green to blue to purple.

```
1317 rainbow_step = 0.005
1318 rainbow_Rgb = 1-rainbow_step -- we start in the red phase
1319 rainbow_rGb = rainbow_step   -- values x must always be 0 < x < 1
1320 rainbow_rgB = rainbow_step
1321 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.

```
1322 randomcolorstring = function()
1323   if randomcolor_grey then
1324     return (0.001*math.random(grey_lower,grey_upper)).." g"
1325   elseif rainbowcolor then
1326     if rainind == 1 then -- red
1327       rainbow_rGb = rainbow_rGb + rainbow_step
1328       if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
1329     elseif rainind == 2 then -- yellow
1330       rainbow_Rgb = rainbow_Rgb - rainbow_step
1331       if rainbow_Rgb <= rainbow_step then rainind = 3 end
1332     elseif rainind == 3 then -- green
1333       rainbow_rgB = rainbow_rgB + rainbow_step
1334       rainbow_rGb = rainbow_rGb - rainbow_step
```

```
1335       if rainbow_rGb <= rainbow_step then rainind = 4 end
1336     elseif rainind == 4 then -- blue
1337       rainbow_Rgb = rainbow_Rgb + rainbow_step
1338       if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
1339     else -- purple
1340       rainbow_rgB = rainbow_rgB - rainbow_step
1341       if rainbow_rgB <= rainbow_step then rainind = 1 end
1342     end
1343     return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rgB.." rg"
1344   else
1345     Rgb = math.random(Rgb_lower,Rgb_upper)/255
1346     rGb = math.random(rGb_lower,rGb_upper)/255
1347     rgB = math.random(rgB_lower,rgB_upper)/255
1348     return Rgb.." "..rGb.." "..rgB.." ".." rg"
1349   end
1350 end
```

11.25.2 randomcolor – the function

The function that does all the colorizing action. It goes through the whole paragraph and looks at every glyph. If the boolean `randomcolor_onlytext` is set, only glyphs with the set attribute will be colored. Elsewise, all glyphs are taken.

```
1351 randomcolor = function(head)
1352   for line in nodetraverseid(0,head) do
1353     for i in nodetraverseid(GLYPH,line.head) do
1354       if not(randomcolor_onlytext) or
1355         (node.has_attribute(i,luatexbase.attributes.randcolorattr))
1356       then
1357         color_push.data = randomcolorstring()  -- color or grey string
1358         line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
1359         nodeinsertafter(line.head,i,nodecopy(color_pop))
1360       end
1361     end
1362   end
1363   return head
1364 end
```

11.26   relationship

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

```
1365   sailheight = 12
1366   mastheight = 4
1367   hullheight = 5
1368   relnumber = 402
1369 function relationship()
```

```
1370 --%% check if there's a problem with any character in the current font
1371   f = font.getfont(font.current())
1372   fullfont = 1
1373   for i = 8756,8842 do
1374     if not(f.characters[i]) then texio.write_nl((i).." not available") fullfont = 0 end
1375   end
1376 --%% store the result of the check for later, then go on to construct the ship:
1377   shipheight = sailheight + mastheight + hullheight
1378   tex.print("\\parshape "..(shipheight)) --%% prepare the paragraph shape ...
1379   for i =1,sailheight do
1380     tex.print(" "..(4.5-i/3.8).."cm "..((i-0.5)/2.5).."cm ")
1381    end
1382   for i =1,mastheight do
1383     tex.print(" "..(3.2).."cm "..(1).."cm ")
1384   end
1385   for i =1,hullheight do
1386     tex.print(" "..((i-1)/2).."cm "..(10-i).."cm ")
1387   end
1388   tex.print("\\noindent") --%% ... up to here, then insert relations
1389   for i=1,relnumber do
1390     tex.print("\\ \\char"..math.random(8756,8842))
1391   end
1392   tex.print("\\break")
1393 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.

```
1394 function cutparagraph(head)
1395   local parsum = 0
1396   for n in nodetraverseid(HLIST,head) do
1397     parsum = parsum + 1
1398     if parsum > shipheight then
1399       node.remove(head,n)
1400     end
1401   end
1402   return head
1403 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.

```
1404 function missingcharstext()
1405   if (fullfont == 0) then
1406   local separator     = string.rep("=", 28)
1407 local texiowrite_nl = texio.write_nl
1408   texiowrite_nl("Output written on "..tex.jobname..".pdf ("..status.total_pages.." chicken,".." e
```

```
1409    texiowrite_nl(" ")
1410    texiowrite_nl(separator)
1411    texiowrite_nl("CAREFUL!!")
1412    texiowrite_nl("\\relationship needs special characters (unicode points 8756 to 8842)")
1413    texiowrite_nl("Your font does not support all of them!")
1414    texiowrite_nl("consider using another one, e.g. the XITS font supplied with TeXlive.")
1415    texiowrite_nl(separator .. "\n")
1416    end
1417 end
```

11.27   rickroll

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

```
1418 %
```

11.28   substitutewords

This function is one of the rather usefull ones of this package. It replaces each occurance 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 TeXs definitions and in Lua. In this second step, the list of substitutions is just extended, and the real work is done by the function `substiuteword` 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.

```
1419 substitutewords_strings = {}
1420
1421 addtosubstitutions = function(input,output)
1422    substitutewords_strings[#substitutewords_strings + 1] = {}
1423    substitutewords_strings[#substitutewords_strings][1] = input
1424    substitutewords_strings[#substitutewords_strings][2] = output
1425 end
1426
1427 substitutewords = function(head)
1428    for i = 1,#substitutewords_strings do
1429       head = string.gsub(head,substitutewords_strings[i][1],substitutewords_strings[i][2])
1430    end
1431    return head
1432 end
```

11.29   suppressonecharbreak

We rush through the node list before line breaking takes place and insert large penalties for breaks after single glyphs. To keep the code as small, simple and fast as possible, we `traverse_id` over spaces and see wether the `next.next` node is also a space. This might not be the best and most universal way of doing

it, but the simplest. The penalty is not created newly each time, but copied – no significant speed gain, however.

```
1433 suppressonecharbreakpenaltynode = node.new(PENALTY)
1434 suppressonecharbreakpenaltynode.penalty = 10000

1435 function suppressonecharbreak(head)
1436   for i in node.traverse_id(GLUE,head) do
1437     if ((i.next) and (i.next.next.id == GLUE)) then
1438         pen = node.copy(suppressonecharbreakpenaltynode)
1439         node.insert_after(head,i.next,pen)
1440     end
1441   end
1442
1443   return head
1444 end
```

11.30 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.

```
1445 tabularasa_onlytext = false
1446
1447 tabularasa = function(head)
1448   local s = nodenew(KERN)
1449   for line in nodetraverseid(HLIST,head) do
1450     for n in nodetraverseid(GLYPH,line.head) do
1451       if not(tabularasa_onlytext) or node.has_attribute(n,luatexbase.attributes.tabularasaattr) t
1452         s.kern = n.width
1453         nodeinsertafter(line.list,n,nodecopy(s))
1454         line.head = noderemove(line.list,n)
1455       end
1456     end
1457   end
1458   return head
1459 end
```

11.31 tanjanize

```
1460 tanjanize = function(head)
1461   local s = nodenew(KERN)
1462   local m = nodenew(GLYPH,1)
1463   local use_letter_i = true
1464   scale = nodenew(WHAT,PDF_LITERAL)
1465   scale2 = nodenew(WHAT,PDF_LITERAL)
1466   scale.data  = "0.5 0 0 0.5 0 0 cm"
1467   scale2.data = "2   0 0 2   0 0 cm"
```

```
1468
1469   for line in nodetraverseid(HLIST,head) do
1470     for n in nodetraverseid(GLYPH,line.head) do
1471       mimicount = 0
1472       tmpwidth  = 0
1473       while ((n.next.id == GLYPH) or (n.next.id == 11) or (n.next.id == 7) or (n.next.id == 0)) d
  find end of a word
1474         n.next = n.next.next
1475         mimicount = mimicount + 1
1476         tmpwidth = tmpwidth + n.width
1477       end
1478
1479     mimi = {}  -- constructing the node list.
1480     mimi[0] = nodenew(GLYPH,1)  -- only a dummy for the loop
1481     for i = 1,string.len(mimicount) do
1482       mimi[i] = nodenew(GLYPH,1)
1483       mimi[i].font = font.current()
1484       if(use_letter_i) then mimi[i].char = 109 else mimi[i].char = 105 end
1485       use_letter_i = not(use_letter_i)
1486       mimi[i-1].next = mimi[i]
1487     end
1488 --]]
1489
1490 line.head = nodeinsertbefore(line.head,n,nodecopy(scale))
1491 nodeinsertafter(line.head,n,nodecopy(scale2))
1492       s.kern = (tmpwidth*2-n.width)
1493       nodeinsertafter(line.head,n,nodecopy(s))
1494     end
1495   end
1496   return head
1497 end
```

11.32 uppercasecolor

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

```
1498 uppercasecolor_onlytext = false
1499
1500 uppercasecolor = function (head)
1501   for line in nodetraverseid(Hhead,head) do
1502     for upper in nodetraverseid(GLYPH,line.head) do
1503       if not(uppercasecolor_onlytext) or node.has_attribute(upper,luatexbase.attributes.uppercase
1504         if (((upper.char > 64) and (upper.char < 91)) or
1505             ((upper.char > 57424) and (upper.char < 57451)))  then  -- for small caps! nice
1506           color_push.data = randomcolorstring()  -- color or grey string
1507           line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
1508           nodeinsertafter(line.head,upper,nodecopy(color_pop))
1509         end
```

```
1510        end
1511      end
1512   end
1513   return head
1514 end
```

11.33   upsidedown

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

```
1515 upsidedown = function(head)
1516   local factor = 65536/0.99626
1517   for line in nodetraverseid(Hhead,head) do
1518     for n in nodetraverseid(GLYPH,line.head) do
1519       if (upsidedownarray[n.char]) then
1520         shift = nodenew(WHAT,PDF_LITERAL)
1521         shift2 = nodenew(WHAT,PDF_LITERAL)
1522         shift.data = "q 1 0 0 -1 0 " .. n.height/factor .." cm"
1523         shift2.data = "Q 1 0 0 1 " .. n.width/factor .." 0 cm"
1524         nodeinsertbefore(head,n,shift)
1525         nodeinsertafter(head,n,shift2)
1526       end
1527     end
1528   end
1529   return head
1530 end
```

11.34   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 ligth 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!

11.34.1   colorstretch – preliminaries

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

```
1531 keeptext = true
1532 colorexpansion = true
1533
1534 colorstretch_coloroffset = 0.5
```

```
1535 colorstretch_colorrange = 0.5
1536 chickenize_rule_bad_height = 4/5 -- height and depth of the rules
1537 chickenize_rule_bad_depth = 1/5
1538
1539
1540 colorstretchnumbers = true
1541 drawstretchthreshold = 0.1
1542 drawexpansionthreshold = 0.9
```

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

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

```
1543 colorstretch = function (head)
1544   local f = font.getfont(font.current()).characters
1545   for line in nodetraverseid(Hhead,head) do
1546     local rule_bad = nodenew(RULE)
1547
1548     if colorexpansion then  -- if also the font expansion should be shown
1549 --%% here use first_glyph function!!
1550       local g = line.head
1551 n = node.first_glyph(line.head.next)
1552 texio.write_nl(line.head.id)
1553 texio.write_nl(line.head.next.id)
1554 texio.write_nl(line.head.next.next.id)
1555 texio.write_nl(n.id)
1556       while not(g.id == GLYPH) and (g.next) do g = g.next end -- find first glyph on line. If line
1557       if (g.id == GLYPH) then                              -- read width only if g is a glyph!
1558         exp_factor = g.expansion_factor/10000 --%% neato, luatex now directly gives me this!!
1559         exp_color = colorstretch_coloroffset + (exp_factor*0.1) .. " g"
1560 texio.write_nl(exp_factor)
1561         rule_bad.width = 0.5*line.width  -- we need two rules on each line!
1562       end
1563     else
1564       rule_bad.width = line.width  -- only the space expansion should be shown, only one rule
1565     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.

```
1566     rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bett
1567     rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth
1568
1569     local glue_ratio = 0
1570     if line.glue_order == 0 then
1571       if line.glue_sign == 1 then
1572         glue_ratio = colorstretch_colorrange * math.min(line.glue_set,1)
```

```
1573        else
1574          glue_ratio = -colorstretch_colorrange * math.min(line.glue_set,1)
1575        end
1576      end
1577      color_push.data = colorstretch_coloroffset + glue_ratio .. " g"
1578
```

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

```
1579 -- set up output
1580      local p = line.head
1581
1582   -- a rule to immitate kerning all the way back
1583      local kern_back = nodenew(RULE)
1584      kern_back.width = -line.width
1585
1586   -- if the text should still be displayed, the color and box nodes are inserted additionally
1587   -- and the head is set to the color node
1588      if keeptext then
1589        line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1590      else
1591        node.flush_list(p)
1592        line.head = nodecopy(color_push)
1593      end
1594      nodeinsertafter(line.head,line.head,rule_bad)  -- then the rule
1595      nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
1596      tmpnode =  nodeinsertafter(line.head,line.head.next.next,kern_back)
1597
1598      -- then a rule with the expansion color
1599      if colorexpansion then  -- if also the stretch/shrink of letters should be shown
1600        color_push.data = exp_color
1601        nodeinsertafter(line.head,tmpnode,nodecopy(color_push))
1602        nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
1603        nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
1604      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.

```
1605      if colorstretchnumbers then
1606        j = 1
1607        glue_ratio_output = {}
1608        for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
1609          local char = unicode.utf8.char(s)
1610          glue_ratio_output[j] = nodenew(GLYPH,1)
1611          glue_ratio_output[j].font = font.current()
1612          glue_ratio_output[j].char = s
```

```
1613          j = j+1
1614       end
1615       if math.abs(glue_ratio) > drawstretchthreshold then
1616          if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
1617          else color_push.data = "0 0.99 0 rg" end
1618       else color_push.data = "0 0 0 rg"
1619       end
1620
1621       nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
1622       for i = 1,math.min(j-1,7) do
1623          nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])
1624       end
1625       nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
1626    end -- end of stretch number insertion
1627  end
1628  return head
1629 end
```

dubstepize

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

```
1630
```

scorpionize

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

```
1631 function scorpionize_color(head)
1632   color_push.data = ".35 .55 .75 rg"
1633   nodeinsertafter(head,head,nodecopy(color_push))
1634   nodeinsertafter(head,node.tail(head),nodecopy(color_pop))
1635   return head
1636 end
```

11.35   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 … (?)

```
1637 substlist = {}
1638 substlist[1488] = 64289
1639 substlist[1491] = 64290
1640 substlist[1492] = 64291
1641 substlist[1499] = 64292
```

```
1642 substlist[1500] = 64293
1643 substlist[1501] = 64294
1644 substlist[1512] = 64295
1645 substlist[1514] = 64296
```

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

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

```
1646 function variantjustification(head)
1647   math.randomseed(1)
1648   for line in nodetraverseid(Hhead,head) do
1649     if (line.glue_sign == 1 and line.glue_order == 0) then -- exclude the last line!
1650       substitutions_wide = {} -- we store all "expandable" letters of each line
1651       for n in nodetraverseid(GLYPH,line.head) do
1652         if (substlist[n.char]) then
1653           substitutions_wide[#substitutions_wide+1] = n
1654         end
1655       end
1656       line.glue_set = 0    -- deactivate normal glue expansion
1657       local width = node.dimensions(line.head)  -- check the new width of the line
1658       local goal = line.width
1659       while (width < goal and #substitutions_wide > 0) do
1660         x = math.random(#substitutions_wide)       -- choose randomly a glyph to be substituted
1661         oldchar = substitutions_wide[x].char
1662         substitutions_wide[x].char = substlist[substitutions_wide[x].char] -- substitute by wide 
1663         width = node.dimensions(line.head)              -- check if the line is too wide
1664         if width > goal then substitutions_wide[x].char = oldchar break end -- substitute back if 
1665         table.remove(substitutions_wide,x)              -- if further substitutions have to be done, 
1666       end
1667     end
1668   end
1669   return head
1670 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.

11.36 zebranize

This function is inspired by a discussion with the Heidelberg regular's table and will change the color of each paragraph linewise. Both the textcolor and background color are changed to create a true zebra like look. If you want to change or add colors, just change the values of zebracolorarray[] for the text colors and zebracolorarray_bg[] for the background. Do not mix with other color changing functions of this package, as that will turn out ugly or erroneous.

The code works just the same as every other thing here: insert color nodes, insert rules, and register the whole thing in post_linebreak_filter.

11.36.1 zebranize – preliminaries

```
1671 zebracolorarray = {}
1672 zebracolorarray_bg = {}
1673 zebracolorarray[1] = "0.1 g"
1674 zebracolorarray[2] = "0.9 g"
1675 zebracolorarray_bg[1] = "0.9 g"
1676 zebracolorarray_bg[2] = "0.1 g"
```

11.36.2 zebranize – the function

This code has to be revisited, it is ugly.

```
1677 function zebranize(head)
1678   zebracolor = 1
1679   for line in nodetraverseid(Hhead,head) do
1680     if zebracolor == #zebracolorarray then zebracolor = 0 end
1681     zebracolor = zebracolor + 1
1682     color_push.data = zebracolorarray[zebracolor]
1683     line.head =     nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1684     for n in nodetraverseid(GLYPH,line.head) do
1685       if n.next then else
1686         nodeinsertafter(line.head,n,nodecopy(color_pull))
1687       end
1688     end
1689
1690     local rule_zebra = nodenew(RULE)
1691     rule_zebra.width = line.width
1692     rule_zebra.height = tex.baselineskip.width*4/5
1693     rule_zebra.depth = tex.baselineskip.width*1/5
1694
1695     local kern_back = nodenew(RULE)
1696     kern_back.width = -line.width
1697
1698     color_push.data = zebracolorarray_bg[zebracolor]
1699     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
1700     line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
1701     nodeinsertafter(line.head,line.head,kern_back)
1702     nodeinsertafter(line.head,line.head,rule_zebra)
1703   end
1704   return (head)
1705 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 sloppinessh and sloppinessv give the amount of sloppiness, i. e. how strongly the points are "wiggled" randomly to make the drawings more dynamically. You can set them at any time in the document

```
1706 --
1707 function pdf_print (...)
1708   for _, str in ipairs({...}) do
1709     pdf.print(str .. " ")
1710   end
1711   pdf.print("\n")
1712 end
1713
1714 function move (p1,p2)
1715   if (p2) then
1716     pdf_print(p1,p2,"m")
1717   else
1718     pdf_print(p1[1],p1[2],"m")
1719   end
1720 end
1721
1722 function line(p1,p2)
1723   if (p2) then
1724     pdf_print(p1,p2,"l")
1725   else
1726     pdf_print(p1[1],p1[2],"l")
1727   end
1728 end
1729
1730 function curve(p11,p12,p21,p22,p31,p32)
1731   if (p22) then
1732     p1,p2,p3 = {p11,p12},{p21,p22},{p31,p32}
1733   else
1734     p1,p2,p3 = p11,p12,p21
1735   end
1736   pdf_print(p1[1], p1[2],
1737             p2[1], p2[2],
```

chicken 62

```
1738                 p3[1], p3[2], "c")
1739 end
1740
1741 function close ()
1742   pdf_print("h")
1743 end
1744
```

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

```
1745 drawwidth = 1
1746
1747 function linewidth (w)
1748   pdf_print(w,"w")
1749 end
1750
1751 function stroke ()
1752   pdf_print("S")
1753 end
1754 --
1755
1756 function strictcircle(center,radius)
1757   local left = {center[1] - radius, center[2]}
1758   local lefttop = {left[1], left[2] + 1.45*radius}
1759   local leftbot = {left[1], left[2] - 1.45*radius}
1760   local right = {center[1] + radius, center[2]}
1761   local righttop = {right[1], right[2] + 1.45*radius}
1762   local rightbot = {right[1], right[2] - 1.45*radius}
1763
1764   move (left)
1765   curve (lefttop, righttop, right)
1766   curve (rightbot, leftbot, left)
1767 stroke()
1768 end
1769
1770 sloppynessh = 5
1771 sloppynessv = 5
1772
1773 function disturb_point(point)
1774   return {point[1] + (math.random() - 1/2)*sloppynessh,
1775          point[2] + (math.random() - 1/2)*sloppynessv}
1776 end
1777
1778 function sloppycircle(center,radius)
1779   local left = disturb_point({center[1] - radius, center[2]})
1780   local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
1781   local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}
```

```lua
1782    local right = disturb_point({center[1] + radius, center[2]})
1783    local righttop = disturb_point({right[1], right[2] + 1.45*radius})
1784    local rightbot = disturb_point({right[1], right[2] - 1.45*radius})
1785
1786    local right_end = disturb_point(right)
1787
1788    move (right)
1789    curve (rightbot, leftbot, left)
1790    curve (lefttop, righttop, right_end)
1791    linewidth(drawwidth*(math.random()+0.5))
1792    stroke()
1793 end
1794
1795 function sloppyellipsis(center,radiusx,radiusy)
1796    local left = disturb_point({center[1] - radiusx, center[2]})
1797    local lefttop = disturb_point({left[1], left[2] + 1.45*radiusy})
1798    local leftbot = {lefttop[1], lefttop[2] - 2.9*radiusy}
1799    local right = disturb_point({center[1] + radiusx, center[2]})
1800    local righttop = disturb_point({right[1], right[2] + 1.45*radiusy})
1801    local rightbot = disturb_point({right[1], right[2] - 1.45*radiusy})
1802
1803    local right_end = disturb_point(right)
1804
1805    move (right)
1806    curve (rightbot, leftbot, left)
1807    curve (lefttop, righttop, right_end)
1808    linewidth(drawwidth*(math.random()+0.5))
1809    stroke()
1810 end
1811
1812 function sloppyline(start,stop)
1813    local start_line = disturb_point(start)
1814    local stop_line = disturb_point(stop)
1815    start = disturb_point(start)
1816    stop = disturb_point(stop)
1817    move(start) curve(start_line,stop_line,stop)
1818    linewidth(drawwidth*(math.random()+0.5))
1819    stroke()
1820 end
```

chicken 64

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:

- LuaTeX 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 LuaTEX 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 …