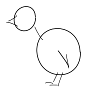

» The Monty Pythons, were they TEX users,
could have written the chickenize macro.«

Paul Isambert

CHICKENIZE

v0.2.9a

Arno L. Trautmann

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: aͤoͤuͤ
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
guttenbergenize	deletes every quote and footnotes
hammertime	U can't touch this!
italianize	Mamma mia!!
italianizerandword	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 chategory 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1 Options for Game of Chicken

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

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

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

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

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

`GOCcellcode` = `<string>` `"scalebox{0.03}{drawchicken}"` The 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 dafault 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 ;)

5.1 How to use a callback

The normal way to use a callback is to "register" a function in it. This way, the function is called each time the callback is executed. Typically, the function takes a node list (see below) as an argument, does something with it, and returns it. So a basic use of the `post_linebreak_filter` would look like:

```
function my_filter(head)
  return head
end

callback.register("post_linebreak_filter",my_filter)
```

The function `callback.register` takes the name of the callback and your new function. However, there are some reasons why we avoid this syntax here. Instead, we rely on the function `luatexbase.add_to_callback`. This is provided by the 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!

6 Nodes

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

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

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

A more convenient way to adress each node of a list is the function `node.traverse(head)` which takes as first argument the first node of the list. However, often one wants to adress only a certain type of nodes in a list – e. g. all glyphs in a vertical list that also contains glue, rules etc. This is achieved by calling

[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.

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.

7 Other things

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

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

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

[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

8 TeX file

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

```
\directlua{dofile(kpse.find_file("chickenize.lua"))}

\def\ALT{%
  \bgroup%
  \fontspec{Latin Modern Sans}%
  A%
  \kern-.375em \raisebox{.65ex}{\scalebox{0.3}{L}}%
  \kern.03em \raisebox{-.99ex}{T}%
  \egroup%
}
```

8.1 allownumberincommands

```
\def\allownumberincommands{
  \catcode`\0=11
  \catcode`\1=11
  \catcode`\2=11
  \catcode`\3=11
  \catcode`\4=11
  \catcode`\5=11
  \catcode`\6=11
  \catcode`\7=11
  \catcode`\8=11
  \catcode`\9=11
}

\def\BEClerize{
  \chickenize
  \directlua{
    chickenstring[1]  = "noise noise"
    chickenstring[2]  = "atom noise"
    chickenstring[3]  = "shot noise"
    chickenstring[4]  = "photon noise"
    chickenstring[5]  = "camera noise"
    chickenstring[6]  = "noising noise"
    chickenstring[7]  = "thermal noise"
    chickenstring[8]  = "electronic noise"
    chickenstring[9]  = "spin noise"
    chickenstring[10] = "electron noise"
    chickenstring[11] = "Bogoliubov noise"
    chickenstring[12] = "white noise"
    chickenstring[13] = "brown noise"
    chickenstring[14] = "pink noise"
    chickenstring[15] = "bloch sphere"
    chickenstring[16] = "atom shot noise"
    chickenstring[17] = "nature physics"
  }
}

\def\boustrophedon{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon,"boustrophedon")}}
\def\unboustrophedon{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon")}}

\def\boustrophedonglyphs{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon_glyphs,"boustrophed
\def\unboustrophedonglyphs{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon_glyphs")}}

\def\boustrophedoninverse{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon_inverse,"boustrophe
\def\unboustrophedoninverse{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon_inverse")}}

\def\bubblesort{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",bubblesort,"bubblesort")}}
\def\unbubblesort{
  \directlua{luatexbase.remove_from_callback("bubblesort","bubblesort")}}

\def\chickenize{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",chickenize,"chickenize")
    luatexbase.add_to_callback("start_page_number",
    function() texio.write("["..status.total_pages) end ,"cstartpage")
    luatexbase.add_to_callback("stop_page_number",
    function() texio.write(" chickens]") end,"cstoppage")
    luatexbase.add_to_callback("stop_run",nicetext,"a nice text")
  }
```

```
}
\def\unchickenize{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","chickenize")
    luatexbase.remove_from_callback("start_page_number","cstartpage")
    luatexbase.remove_from_callback("stop_page_number","cstoppage")}}

\def\coffeestainize{  %% to be implemented.
  \directlua{}}
\def\uncoffeestainize{
  \directlua{}}

\def\colorstretch{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",colorstretch,"stretch_expansion")}
\def\uncolorstretch{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","stretch_expansion")}}

\def\countglyphs{
  \directlua{
             counted_glyphs_by_code = {}
             for i = 1,10000 do
               counted_glyphs_by_code[i] = 0
             end
             glyphnumber = 0 spacenumber = 0
             luatexbase.add_to_callback("post_linebreak_filter",countglyphs,"countglyphs")
             luatexbase.add_to_callback("stop_run",printglyphnumber,"printglyphnumber")
  }
}

\def\countwords{
  \directlua{wordnumber = 0
             luatexbase.add_to_callback("pre_linebreak_filter",countwords,"countwords")
             luatexbase.add_to_callback("stop_run",printwordnumber,"printwordnumber")
  }
}

\def\detectdoublewords{
  \directlua{
             luatexbase.add_to_callback("post_linebreak_filter",detectdoublewords,"detectdoublewo
             luatexbase.add_to_callback("stop_run",printdoublewords,"printdoublewords")
  }
}

\def\dosomethingfunny{
    %% should execute one of the "funny" commands, but randomly. So every compilation is complete
functions. Maybe also on a per-paragraph-basis?
}
```

```

\def\dubstepenize{
  \chickenize
  \directlua{
    chickenstring[1] = "WOB"
    chickenstring[2] = "WOB"
    chickenstring[3] = "WOB"
    chickenstring[4] = "BROOOAR"
    chickenstring[5] = "WHEE"
    chickenstring[6] = "WOB WOB WOB"
    chickenstring[7] = "WAAAAAAAAH"
    chickenstring[8] = "duhduh duhduh duh"
    chickenstring[9] = "BEEEEEEEEEW"
    chickenstring[10] = "DDEEEEEEEW"
    chickenstring[11] = "EEEEEW"
    chickenstring[12] = "boop"
    chickenstring[13] = "buhdee"
    chickenstring[14] = "bee bee"
    chickenstring[15] = "BZZZRRRRRRROOOOOOAAAAA"

    chickenizefraction = 1
  }
}
\let\dubstepize\dubstepenize

\def\explainbackslashes{ %% inspired by xkcd #1638
  {\tt\noindent
\textbackslash escape character\\
\textbackslash\textbackslash line end or escaped escape character in tex.print("")\\
\textbackslash\textbackslash\textbackslash real, real backslash\\
\textbackslash\textbackslash\textbackslash\textbackslash line end in tex.print("")\\
\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash elder backslash \\
\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash backslash whi
\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslas
\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslas
\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslash\textbackslas
  eater}
}

\def\francize{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",francize,"francize")}}

\def\unfrancize{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter",francize)}}

\def\gameoflife{
  Your Life Is Tetris. Stop Playing It Like Chess.
}
```

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!

```
\def\gameofchicken#1{\directlua{
GOCrule_live = {2,3}
GOCrule_spawn = {3}
GOCrule_antilive = {2,3}
GOCrule_antispawn = {3}
GOCcellcode = "\\scalebox{0.03}{\\drawchicken}"
GOCcellcode = "\\scalebox{0.03}{\\drawcov}"
GOCx = 100
GOCy = 100
GOCiter = 150
GOC_console = false
GOC_pdf = true
GOCsleep = 0
GOCdensity = 100
#1
gameofchicken()

if (GOCmakegif == true) then
  luatexbase.add_to_callback("wrapup_run",make_a_gif,"makeagif")
end
}}
\let\gameofchimken\gameofchicken % yeah, that had to be.

\def\guttenbergenize{ %% makes only sense when using LaTeX
  \AtBeginDocument{
    \let\grqq\relax\let\glqq\relax
    \let\frqq\relax\let\flqq\relax
    \let\grq\relax\let\glq\relax
    \let\frq\relax\let\flq\relax
%
    \gdef\footnote##1{}
    \gdef\cite##1{}\gdef\parencite##1{}
    \gdef\Cite##1{}\gdef\Parencite##1{}
    \gdef\cites##1{}\gdef\parencites##1{}
    \gdef\Cites##1{}\gdef\Parencites##1{}
    \gdef\footcite##1{}\gdef\footcitetext##1{}
    \gdef\footcites##1{}\gdef\footcitetexts##1{}
    \gdef\textcite##1{}\gdef\Textcite##1{}
    \gdef\textcites##1{}\gdef\Textcites##1{}
    \gdef\smartcites##1{}\gdef\Smartcites##1{}
    \gdef\supercite##1{}\gdef\supercites##1{}
    \gdef\autocite##1{}\gdef\Autocite##1{}
    \gdef\autocites##1{}\gdef\Autocites##1{}
    %% many, many missing … maybe we need to tackle the underlying mechanism?
  }
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",guttenbergenize_rq,"guttenbergenize
}

\def\hammertime{
  \global\let\n\relax
  \directlua{hammerfirst = true
             luatexbase.add_to_callback("pre_linebreak_filter",hammertime,"hammertime")}}
\def\unhammertime{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","hammertime")}}

\let\hendlnize\chickenize     % homage to Hendl/Chicken
\let\unhendlnize\unchickenize % may the soldering strength always be with him

\def\italianizerandword{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",italianizerandword,"italianizerandw
\def\unitalianizerandword{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","italianizerandword")}}

\def\italianize{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",italianize,"italianize")}}
\def\unitalianize{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","italianize")}}

% \def\itsame{
%   \directlua{drawmario}} %%% does not exist

\def\kernmanipulate{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",kernmanipulate,"kernmanipulate")}}
\def\unkernmanipulate{
  \directlua{lutaexbase.remove_from_callback("pre_linebreak_filter",kernmanipulate)}}

\def\leetspeak{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",leet,"1337")}}
\def\unleetspeak{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","1337")}}

\def\leftsideright#1{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",leftsideright,"leftsideright")}
  \directlua{
    leftsiderightindex = {#1}
    leftsiderightarray = {}
    for _,i in pairs(leftsiderightindex) do
      leftsiderightarray[i] = true
    end
  }
}
\def\unleftsideright{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","leftsideright")}}

\def\letterspaceadjust{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",letterspaceadjust,"letterspaceadju
\def\unletterspaceadjust{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","letterspaceadjust")}}

\def\listallcommands{
  \directlua{
 for name in pairs(tex.hashtokens()) do
     print(name)
 end}
}

\let\stealsheep\letterspaceadjust     %% synonym in honor of Paul
\let\unstealsheep\unletterspaceadjust
\let\returnsheep\unletterspaceadjust

\def\matrixize{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",matrixize,"matrixize")}}
\def\unmatrixize{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","matrixize")}}

\def\milkcow{     %% FIXME %% to be implemented
  \directlua{}}
\def\unmilkcow{
  \directlua{}}

\def\medievalumlaut{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",medievalumlaut,"medievalumlaut")}
\def\unmedievalumlaut{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","medievalumlaut")}}

\def\pancakenize{
  \directlua{luatexbase.add_to_callback("stop_run",pancaketext,"pancaketext")}}

\def\rainbowcolor{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"rainbowcolor")
             rainbowcolor = true}}
\def\unrainbowcolor{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","rainbowcolor")
             rainbowcolor = false}}
```

```
\let\nyanize\rainbowcolor
\let\unnyanize\unrainbowcolor

\def\randomchars{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomchars,"randomchars")}}
\def\unrandomchars{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomchars")}}

\def\randomcolor{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomcolor,"randomcolor")}}
\def\unrandomcolor{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomcolor")}}

\def\randomerror{ %% FIXME
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomerror,"randomerror")}}
\def\unrandomerror{ %% FIXME
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomerror")}}

\def\randomfonts{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomfonts,"randomfonts")}}
\def\unrandomfonts{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","randomfonts")}}

\def\randomuclc{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",randomuclc,"randomuclc")}}
\def\unrandomuclc{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","randomuclc")}}

\def\relationship{%
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",cutparagraph,"cut paragraph")
    relationship()
  }
}

\let\rongorongonize\boustrophedoninverse
\let\unrongorongonize\unboustrophedoninverse

\def\scorpionize{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",scorpionize_color,"scorpionize_col
\def\unscorpionize{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","scorpionize_color")}}

\def\spankmonkey{    %% to be implemented
  \directlua{}}
\def\unspankmonkey{
  \directlua{}}
```

```

\def\substitutewords{
  \directlua{luatexbase.add_to_callback("process_input_buffer",substitutewords,"substitutewords")}
\def\unsubstitutewords{
  \directlua{luatexbase.remove_from_callback("process_input_buffer","substitutewords")}}

\def\addtosubstitutions#1#2{
  \directlua{addtosubstitutions("#1","#2")}
}

\def\suppressonecharbreak{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",suppressonecharbreak,"suppressonec
\def\unsuppressonecharbreak{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","suppressonecharbreak")}}

\def\tabularasa{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",tabularasa,"tabularasa")}}
\def\untabularasa{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","tabularasa")}}

\def\tanjanize{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",tanjanize,"tanjanize")}}
\def\untanjanize{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","tanjanize")}}

\def\uppercasecolor{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",uppercasecolor,"uppercasecolor")}
\def\unuppercasecolor{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","uppercasecolor")}}

\def\upsidedown#1{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",upsidedown,"upsidedown")}
  \directlua{
    upsidedownindex = {#1}
    upsidedownarray = {}
    for _,i in pairs(upsidedownindex) do
      upsidedownarray[i] = true
    end
  }
}
\def\unupsidedown{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","upsidedown")}}

\def\variantjustification{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",variantjustification,"variantjusti
\def\unvariantjustification{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","variantjustification")}}

\def\zebranize{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",zebranize,"zebranize")}}
\def\unzebranize{
  \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.

```
\newattribute\leetattr
\newattribute\letterspaceadjustattr
\newattribute\randcolorattr
\newattribute\randfontsattr
\newattribute\randuclcattr
\newattribute\tabularasaattr
\newattribute\uppercasecolorattr

\long\def\textleetspeak#1%
  {\setluatexattribute\leetattr{42}#1\unsetluatexattribute\leetattr}

\long\def\textletterspaceadjust#1{
  \setluatexattribute\letterspaceadjustattr{42}#1\unsetluatexattribute\letterspaceadjustattr
  \directlua{
    if (textletterspaceadjustactive) then else % -- if already active, do nothing
      luatexbase.add_to_callback("pre_linebreak_filter",textletterspaceadjust,"textletterspaceadj
    end
    textletterspaceadjustactive = true         % -- set to active
  }
}
\let\textlsa\textletterspaceadjust

\long\def\textrandomcolor#1%
  {\setluatexattribute\randcolorattr{42}#1\unsetluatexattribute\randcolorattr}
\long\def\textrandomfonts#1%
  {\setluatexattribute\randfontsattr{42}#1\unsetluatexattribute\randfontsattr}
\long\def\textrandomfonts#1%
  {\setluatexattribute\randfontsattr{42}#1\unsetluatexattribute\randfontsattr}
\long\def\textrandomuclc#1%
  {\setluatexattribute\randuclcattr{42}#1\unsetluatexattribute\randuclcattr}
\long\def\texttabularasa#1%
  {\setluatexattribute\tabularasaattr{42}#1\unsetluatexattribute\tabularasaattr}
\long\def\textuppercasecolor#1%
  {\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.

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

8.2 drawchicken

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

```
\long\def\luadraw#1#2{%
  \vbox to #1bp{%
    \vfil
    \latelua{pdf_print("q") #2 pdf_print("Q")}%
  }%
}
\long\def\drawchicken{
  \luadraw{90}{
    chickenhead     = {200,50} % chicken head center
    chickenhead_rad = 20

    neckstart = {215,35} % neck
    neckstop  = {230,10} %

    chickenbody     = {260,-10}
    chickenbody_rad = 40
    chickenleg = {
      {{260,-50},{250,-70},{235,-70}},
      {{270,-50},{260,-75},{245,-75}}
    }

    beak_top = {185,55}
    beak_front = {165,45}
    beak_bottom = {185,35}

    wing_front = {260,-10}
    wing_bottom = {280,-40}
    wing_back = {275,-15}

    sloppycircle(chickenhead,chickenhead_rad) sloppyline(neckstart,neckstop)
    sloppycircle(chickenbody,chickenbody_rad)
    sloppyline(chickenleg[1][1],chickenleg[1][2]) sloppyline(chickenleg[1][2],chickenleg[1][3])
    sloppyline(chickenleg[2][1],chickenleg[2][2]) sloppyline(chickenleg[2][2],chickenleg[2][3])
    sloppyline(beak_front,beak_top) sloppyline(beak_front,beak_bottom)
    sloppyline(wing_front,wing_bottom) sloppyline(wing_back,wing_bottom)
  }
}
```

8.3 drawcov

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

```
\long\def\drawcov{
  \luadraw{90}{
    covbody = {200,50}
    covbody_rad = 50

    covcrown_rad = 5
    crownno = 13
    for i=1,crownno do
      crownpos = {covbody[1]+1.4*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+1.4*covbody_
      crownconnect = {covbody[1]+covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+covbody_rad
     sloppycircle(crownpos,covcrown_rad)
     sloppyline(crownpos,crownconnect)
   end

    covcrown_rad = 6
    crownno = 8
    for i=1,crownno do
      crownpos = {covbody[1]+0.8*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+0.8*covbody_
      crownconnect = {covbody[1]+0.5*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+0.5*covb
      sloppycircle(crownpos,covcrown_rad)
      sloppyline(crownpos,crownconnect)
    end

    covcrown_rad = 8
    sloppycircle(covbody,covcrown_rad)
    sloppycircle(covbody,covbody_rad)
    sloppyline(covbody,covbody)
  }
}
```

9 LaTeX package

I have decided to keep the LaTeX-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.

```
\ProvidesPackage{chickenize}%
  [2020/05/02 v0.2.9a chickenize package]
\input{chickenize}
```

9.1 Free Compliments

```
%
```

9.2 Definition of User-Level Macros

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

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

10 Lua Module

This file contains all the necessary functions and is the actual work horse of this package. The functions are sorted alphabetically (or, they *should* be …) and not by sense, functionality or anything.

First, we set up some constants that are used by many of the following functions. These are made global so the code can be manipulated at the document level, too.

```

local nodeid   = node.id
local nodecopy = node.copy
local nodenew  = node.new
local nodetail = node.tail
local nodeslide = node.slide
local noderemove = node.remove
local nodetraverseid = node.traverse_id
local nodeinsertafter = node.insert_after
local nodeinsertbefore = node.insert_before

Hhead = nodeid("hhead")
RULE  = nodeid("rule")
GLUE  = nodeid("glue")
WHAT  = nodeid("whatsit")
COL   = node.subtype("pdf_colorstack")
DISC  = nodeid("disc")
GLYPH = nodeid("glyph")
GLUE  = nodeid("glue")
HLIST = nodeid("hlist")
KERN  = nodeid("kern")
PUNCT = nodeid("punct")
PENALTY = nodeid("penalty")
```

```
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`.

```
color_push = nodenew(WHAT,COL)
color_pop = nodenew(WHAT,COL)
color_push.stack = 0
color_pop.stack = 0
color_push.command = 1
color_pop.command = 2
```

10.1 chickenize

The infamous `\chickenize` macro. Substitutes every word of the input with the given string. This can be elaborated arbitrarily, and whenever I feel like, I might add functionality. So far, only the string replaces the word, and even hyphenation is not possible.

```
chicken_pagenumbers = true

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

chickenizefraction = 0.5 -- set this to a small value to fool somebody, or to see if your text ha
chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing you

local match = unicode.utf8.match
chickenize_ignore_word = false
```

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

```
chickenize_real_stuff = function(i,head)
    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
      i.next = i.next.next
    end

    chicken = {}  -- constructing the node list.

-- Should this be done only once? No, otherwise we lose the freedom to change the string in-document.
-- But it could be done only once each paragraph as in-paragraph changes are not possible!

    chickenstring_tmp = chickenstring[math.random(1,#chickenstring)]
    chicken[0] = nodenew(GLYPH,1)  -- only a dummy for the loop
    for i = 1,string.len(chickenstring_tmp) do
      chicken[i] = nodenew(GLYPH,1)
      chicken[i].font = font.current()
      chicken[i-1].next = chicken[i]
    end

    j = 1
    for s in string.utfvalues(chickenstring_tmp) do
      local char = unicode.utf8.char(s)
      chicken[j].char = s
      if match(char,"%s") then
        chicken[j] = nodenew(GLUE)
        chicken[j].width = space
        chicken[j].shrink = shrink
        chicken[j].stretch = stretch
      end
      j = j+1
    end

    nodeslide(chicken[1])
    lang.hyphenate(chicken[1])
    chicken[1] = node.kerning(chicken[1])    -- FIXME: does not work
    chicken[1] = node.ligaturing(chicken[1]) -- dito

    nodeinsertbefore(head,i,chicken[1])
    chicken[1].next = chicken[2] -- seems to be necessary … to be fixed
    chicken[string.len(chickenstring_tmp)].next = i.next

    -- shift lowercase latin letter to uppercase if the original input was an uppercase
    if (chickenize_capital and (chicken[1].char > 96 and chicken[1].char < 123)) then
      chicken[1].char = chicken[1].char - 32
    end

  return head
end

chickenize = function(head)
  for i in nodetraverseid(GLYPH,head) do  --find start of a word
    -- Random determination of the chickenization of the next word:
    if math.random() > chickenizefraction then
      chickenize_ignore_word = true
    elseif chickencount then
      chicken_substitutions = chicken_substitutions + 1
    end

    if (chickenize_ignore_word == false) then  -- normal case: at the beginning of a word, we jum
      if (i.char > 64 and i.char < 91) then chickenize_capital = true else chickenize_capital = f
      head = chickenize_real_stuff(i,head)
    end

-- At the end of the word, the ignoring is reset. New chance for everyone.
    if not((i.next.id == GLYPH) or (i.next.id == DISC) or (i.next.id == PUNCT) or (i.next.id == K
      chickenize_ignore_word = false
    end
  end
  return head
end

```

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)

```
local separator     = string.rep("=", 28)
local texiowrite_nl = texio.write_nl
nicetext = function()
  texiowrite_nl("Output written on "..tex.jobname..".pdf ("..status.total_pages.." chicken,".." eg
  texiowrite_nl(" ")
  texiowrite_nl(separator)
  texiowrite_nl("Hello my dear user,")
  texiowrite_nl("good job, now go outside and enjoy the world!")
  texiowrite_nl(" ")
  texiowrite_nl("And don't forget to feed your chicken!")
  texiowrite_nl(separator .. "\n")
  if chickencount then
    texiowrite_nl("There were "..chicken_substitutions.." substitutions made.")
    texiowrite_nl(separator)
  end
end
```

10.2 boustrophedon

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

Linewise rotation:

```
boustrophedon = function(head)
  rot = node.new(WHAT,PDF_LITERAL)
  rot2 = node.new(WHAT,PDF_LITERAL)
  odd = true
    for line in node.traverse_id(0,head) do
      if odd == false then
        w = line.width/65536*0.99625 -- empirical correction factor (?)
        rot.data  = "-1 0 0 1 "..w.." 0 cm"
        rot2.data = "-1 0 0 1 "..-w.." 0 cm"
        line.head = node.insert_before(line.head,line.head,nodecopy(rot))
        nodeinsertafter(line.head,nodetail(line.head),nodecopy(rot2))
        odd = true
      else
        odd = false
      end
    end
  return head
end
```

Glyphwise rotation:

```
boustrophedon_glyphs = function(head)
  odd = false
  rot = nodenew(WHAT,PDF_LITERAL)
  rot2 = nodenew(WHAT,PDF_LITERAL)
  for line in nodetraverseid(0,head) do
    if odd==true then
      line.dir = "TRT"
      for g in nodetraverseid(GLYPH,line.head) do
        w = -g.width/65536*0.99625
        rot.data = "-1 0 0 1 " .. w .." 0 cm"
        rot2.data = "-1 0 0 1 " .. -w .." 0 cm"
        line.head = node.insert_before(line.head,g,nodecopy(rot))
        nodeinsertafter(line.head,g,nodecopy(rot2))
      end
      odd = false
      else
        line.dir = "TLT"
        odd = true
      end
    end
  return head
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.

```
boustrophedon_inverse = function(head)
  rot = node.new(WHAT,PDF_LITERAL)
  rot2 = node.new(WHAT,PDF_LITERAL)
  odd = true
    for line in node.traverse_id(0,head) do
      if odd == false then
texio.write_nl(line.height)
        w = line.width/65536*0.99625 -- empirical correction factor (?)
        h = line.height/65536*0.99625
        rot.data  = "-1 0 0 -1 "..w.." "..h.." cm"
        rot2.data = "-1 0 0 -1 "..-w.." "..0.5*h.." cm"
        line.head = node.insert_before(line.head,line.head,node.copy(rot))
        node.insert_after(line.head,node.tail(line.head),node.copy(rot2))
        odd = true
      else
        odd = false
      end
    end
  return head
end
```

10.3 bubblesort

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

```
function bubblesort(head)
  for line in nodetraverseid(0,head) do
    for glyph in nodetraverseid(GLYPH,line.head) do

    end
  end
  return head
end
```

10.4 countglyphs

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

```
countglyphs = function(head)
  for line in nodetraverseid(0,head) do
    for glyph in nodetraverseid(GLYPH,line.head) do
      glyphnumber = glyphnumber + 1
      if (glyph.next.next) then
        if (glyph.next.id == 10) and (glyph.next.next.id == GLYPH) then
          spacenumber = spacenumber + 1
        end
        counted_glyphs_by_code[glyph.char] = counted_glyphs_by_code[glyph.char] + 1
      end
    end
  end
  return head
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? …

```
printglyphnumber = function()
  texiowrite_nl("\nNumber of glyphs by character code (only up to 127):")
  for i = 1,127 do --%% FIXME: should allow for more characters, but cannot be printed to console
    texiowrite_nl(string.char(i)..": "..counted_glyphs_by_code[i])
  end

  texiowrite_nl("\nTotal number of glyphs in this document: "..glyphnumber)
  texiowrite_nl("Number of spaces in this document: "..spacenumber)
  texiowrite_nl("Glyphs plus spaces: "..glyphnumber+spacenumber.."\n")
end
```

10.5 countwords

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

```
countwords = function(head)
  for glyph in nodetraverseid(GLYPH,head) do
    if (glyph.next.id == 10) then
      wordnumber = wordnumber + 1
    end
  end
  wordnumber = wordnumber + 1 -- add 1 for the last word in a paragraph which is not found otherwi
  return head
end
```

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

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

10.6 detectdoublewords

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

```
      newfirstword[#newfirstword+1] = g.char
      if (g.next.id == 10) then break end
    end
texio.write_nl("nfw:"..#newfirstword)
  end
end

printdoublewords = function()
  texio.write_nl("finished")
end
```

10.7 francize

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

```
francize = function(head)
  for n in nodetraverseid(GLYPH,head) do
    if ((n.char > 47) and (n.char < 58)) then
      n.char = math.random(48,57)
    end
  end
  return head
end
```

10.8 gamofchicken

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

```
function gameofchicken()
  GOC_lifetab = {}
  GOC_spawntab = {}
  GOC_antilifetab = {}
  GOC_antispawntab = {}
  -- translate the rules into an easily-manageable table
  for i=1,#GOCrule_live do; GOC_lifetab[GOCrule_live[i]] = true end
  for i=1,#GOCrule_spawn do; GOC_spawntab[GOCrule_spawn[i]] = true end
  for i=1,#GOCrule_antilive do; GOC_antilifetab[GOCrule_antilive[i]] = true end
  for i=1,#GOCrule_antispawn do; GOC_antispawntab[GOCrule_antispawn[i]] = true end
```

Initialize the arrays for cells and anticells with zeros.

```
-- initialize the arrays
local life = {}
local antilife = {}
local newlife = {}
local newantilife = {}
for i = 0, GOCx do life[i] = {}; newlife[i] = {} for j = 0, GOCy do life[i][j] = 0 end end
for i = 0, GOCx do antilife[i] = {}; newantilife[i] = {} for j = 0, GOCy do antilife[i][j] = 0 enc
```

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

```
function applyruleslife(neighbors, lifeij, antineighbors, antilifeij)
  if GOC_spawntab[neighbors] then myret = 1 else -- new cell
  if GOC_lifetab[neighbors] and (lifeij == 1) then myret = 1 else myret =  0 end end
  if antineighbors > 1 then myret =  0 end
  return myret
end
function applyrulesantilife(neighbors, lifeij, antineighbors, antilifeij)
  if (antineighbors == 3) then myret = 1 else -- new cell or keep cell
  if (((antineighbors > 1) and (antineighbors < 4)) and (lifeij == 1)) then myret = 1 else myret =
  if neighbors > 1 then myret =  0 end
  return myret
end
```

Preparing the initial state with a default pattern:

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

```
  print("start");
  for i = 1,GOCx do
    for j = 1,GOCy do
      if (life[i][j]==1) then texio.write("X") else if (antilife[i][j]==1) then texio.write("O") e
    end
    texio.write_nl(" ");
  end
  os.sleep(GOCsleep)

  for i = 0, GOCx do
    for j = 0, GOCy do
        newlife[i][j] = 0 -- Fill the values from the start settings here
        newantilife[i][j] = 0 -- Fill the values from the start settings here
    end
  end

  for k = 1,GOCiter do -- iterate over the cycles
    texio.write_nl(k);
    for i = 1, GOCx-1 do -- iterate over lines
      for j = 1, GOCy-1 do -- iterate over columns -- prevent edge effects
        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])
        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

        newlife[i][j] = applyruleslife(neighbors, life[i][j],antineighbors, antilife[i][j])
        newantilife[i][j] = applyrulesantilife(neighbors,life[i][j], antineighbors,antilife[i][j]
      end
    end

    for i = 1, GOCx do
      for j = 1, GOCy do
        life[i][j] = newlife[i][j] -- copy the values
        antilife[i][j] = newantilife[i][j] -- copy the values
      end
    end

    for i = 1,GOCx do
      for j = 1,GOCy do
        if GOC_console then
          if (life[i][j]==1) then texio.write("X") else if (antilife[i][j]==1) then texio.write("
        end
        if GOC_pdf then
          if (life[i][j]==1) then tex.print("\\placeat("..(i/10)..","..(j/10).."){"..GOCcellcode.
          if (antilife[i][j]==1) then tex.print("\\placeat("..(i/10)..","..(j/10).."){"..GOCantic
        end
      end
    end
    tex.print(".\\newpage")
    os.sleep(GOCsleep)
  end
 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.

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

10.9 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

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

about the real sources of your work.

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

10.9.1 guttenbergenize – preliminaries

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

```
local quotestrings = {
   [171] = true,  [172] = true,
  [8216] = true, [8217] = true, [8218] = true,
  [8219] = true, [8220] = true, [8221] = true,
  [8222] = true, [8223] = true,
  [8248] = true, [8249] = true, [8250] = true,
}
```

10.9.2 guttenbergenize – the function

```
guttenbergenize_rq = function(head)
  for n in nodetraverseid(GLYPH,head) do
    local i = n.char
    if quotestrings[i] then
      noderemove(head,n)
    end
  end
  return head
end
```

10.10 hammertime

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

```
hammertimedelay = 1.2
local htime_separator = string.rep("=", 30) .. "\n" -- slightly inconsistent with the "nicetext"
hammertime = function(head)
  if hammerfirst then
    texiowrite_nl(htime_separator)
    texiowrite_nl("============STOP!=============\n")
    texiowrite_nl(htime_separator .. "\n\n\n")
    os.sleep (hammertimedelay*1.5)
    texiowrite_nl(htime_separator .. "\n")
    texiowrite_nl("==========HAMMERTIME==========\n")
    texiowrite_nl(htime_separator .. "\n\n")
    os.sleep (hammertimedelay)
```

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

```
    hammerfirst = false
  else
    os.sleep (hammertimedelay)
    texiowrite_nl(htime_separator)
    texiowrite_nl("======U can't touch this!=====\n")
    texiowrite_nl(htime_separator .. "\n\n")
    os.sleep (hammertimedelay*0.5)
  end
  return head
end
```

10.11 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.

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

italianize = function(head)
  -- skip "h/H" randomly
  for n in node.traverse_id(GLYPH,head) do -- go through all glyphs
      if n.prev.id ~= GLYPH then -- check if it's a word start
      if ((n.char == 72) or (n.char == 104)) and (tex.normal_rand() < italianizefraction) then --
        n.prev.next = n.next
      end
    end
  end

  -- add h or H in front of vowels
  for n in nodetraverseid(GLYPH,head) do
    if math.random() < italianizefraction then
    x = n.char
    if x == 97 or x == 101 or x == 105 or x == 111 or x == 117 or
       x == 65 or x ==  69 or x ==  73 or x == 79 or x == 85 then
      if (n.prev.id == GLUE) then
        mynode.font = n.font
        if x > 90 then  -- lower case
          mynode.char = 104
        else
          mynode.char = 72 -- upper case - convert into lower case
          n.char = x + 32
        end
          node.insert_before(head,n,node.copy(mynode))
        end
      end
    end
  end

  -- add e after words, but only after consonants
  for n in node.traverse_id(GLUE,head) do
    if n.prev.id == GLYPH then
    x = n.prev.char
    -- skip vowels and randomize
    if not(x == 97 or x == 101 or x == 105 or x == 111 or x == 117 or x == 44 or x == 46) and mat
        mynode.char = 101           -- it's always a lower case e, no?
        mynode.font = n.prev.font -- adapt the current font
        node.insert_before(head,n,node.copy(mynode)) -- insert the e in the node list
      end
    end
  end

  return head
end
% \subsection{italianize}\label{sec:italianizerandword}
% This is inspired by my dearest colleagues and their artistic interpretation of the english gram
%    \begin{macrocode}
italianizerandwords = function(head)
words = {}
-- head.next.next is the very first word. However, let's try to get the first word after the firs
wordnumber = 0
  for n in nodetraverseid(GLUE,head) do -- let's try to count words by their separators
    wordnumber = wordnumber + 1
    if n.next then
      texio.write_nl(n.next.char)
      words[wordnumber] = {}
      words[wordnumber][1] = node.copy(n.next)

      glyphnumber = 1
      myglyph = n.next
        while myglyph.next do
          node.tail(words[wordnumber][1]).next = node.copy(myglyph.next)
          myglyph = myglyph.next
        end
      end
    end
myinsertnode = head.next.next -- first letter
node.tail(words[1][1]).next = myinsertnode.next
myinsertnode.next = words[1][1]

  return head
```

```
end

italianize_old = function(head)
  local wordlist = {} -- here we will store the number of words of the sentence.
  local words = {} -- here we will store the words of the sentence.
  local wordnumber = 0
  -- let's first count all words in one sentence, howboutdat?
  wordlist[wordnumber] = 1 -- let's save the word *length* in here …


  for n in nodetraverseid(GLYPH,head) do
    if (n.next.id == GLUE) then -- this is a space
      wordnumber = wordnumber + 1
      wordlist[wordnumber] = 1
      words[wordnumber] = n.next.next
    end
    if (n.next.id == GLYPH) then  -- it's a glyph
    if (n.next.char == 46) then -- this is a full stop.
      wordnumber = wordnumber + 1
      texio.write_nl("this sentence had "..wordnumber.."words.")
      for i=0,wordnumber-1 do
      texio.write_nl("word "..i.." had " .. wordlist[i] .. "glyphs")
      end
      texio.write_nl(" ")
      wordnumber = -1 -- to compensate the fact that the next node will be a space, this would co
    else

      wordlist[wordnumber] = wordlist[wordnumber] + 1 -- the current word got 1 glyph longer
      end
    end
  end
  return head
end
```

10.12 hammertime

This is a completely useless function. It just prints STOP! – HAMMERTIME at the beginnig of the first paragraph after `\hammertime`, and "U can't touch this" for every following one. As the function writes to the terminal, you have to be sure that your terminal is line-buffered and not block-buffered. Compare the explanation by Taco on the LuaTEX mailing list.[13]

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

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

```
    texiowrite_nl(htime_separator)
    texiowrite_nl("============STOP!=============\n")
    texiowrite_nl(htime_separator .. "\n\n\n")
    os.sleep (hammertimedelay*1.5)
    texiowrite_nl(htime_separator .. "\n")
    texiowrite_nl("==========HAMMERTIME==========\n")
    texiowrite_nl(htime_separator .. "\n\n")
    os.sleep (hammertimedelay)
    hammerfirst = false
  else
    os.sleep (hammertimedelay)
    texiowrite_nl(htime_separator)
    texiowrite_nl("======U can't touch this!=====\n")
    texiowrite_nl(htime_separator .. "\n\n")
    os.sleep (hammertimedelay*0.5)
  end
  return head
end
```

10.13 itsame

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

```
itsame = function()
local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
color = "1 .6 0"
for i = 6,9 do mr(i,3) end
for i = 3,11 do mr(i,4) end
for i = 3,12 do mr(i,5) end
for i = 4,8 do mr(i,6) end
for i = 4,10 do mr(i,7) end
for i = 1,12 do mr(i,11) end
for i = 1,12 do mr(i,12) end
for i = 1,12 do mr(i,13) end

color = ".3 .5 .2"
for i = 3,5 do mr(i,3) end mr(8,3)
mr(2,4) mr(4,4) mr(8,4)
mr(2,5) mr(4,5) mr(5,5) mr(9,5)
mr(2,6) mr(3,6) for i = 8,11 do mr(i,6) end
for i = 3,8 do mr(i,8) end
for i = 2,11 do mr(i,9) end
for i = 1,12 do mr(i,10) end
mr(3,11) mr(10,11)
for i = 2,4 do mr(i,15) end for i = 9,11 do mr(i,15) end
for i = 1,4 do mr(i,16) end for i = 9,12 do mr(i,16) end
```

```

color = "1 0 0"
for i = 4,9 do mr(i,1) end
for i = 3,12 do mr(i,2) end
for i = 8,10 do mr(5,i) end
for i = 5,8 do mr(i,10) end
mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
for i = 4,9 do mr(i,12) end
for i = 3,10 do mr(i,13) end
for i = 3,5 do mr(i,14) end
for i = 7,10 do mr(i,14) end
end
```

10.14 kernmanipulate

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

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

```
chickenkernamount = 0
chickeninvertkerning = false

function kernmanipulate (head)
  if chickeninvertkerning then -- invert the kerning
    for n in nodetraverseid(11,head) do
      n.kern = -n.kern
    end
  else             -- if not, set it to the given value
    for n in nodetraverseid(11,head) do
      n.kern = chickenkernamount
    end
  end
  return head
end
```

10.15 leetspeak

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

```
leetspeak_onlytext = false
leettable = {
  [101] = 51, -- E
  [105] = 49, -- I
  [108] = 49, -- L
  [111] = 48, -- O
  [115] = 53, -- S
  [116] = 55, -- T

  [101-32] = 51, -- e
  [105-32] = 49, -- i
  [108-32] = 49, -- l
  [111-32] = 48, -- o
  [115-32] = 53, -- s
  [116-32] = 55, -- t
}
```

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

```
leet = function(head)
  for line in nodetraverseid(Hhead,head) do
    for i in nodetraverseid(GLYPH,line.head) do
      if not leetspeak_onlytext or
         node.has_attribute(i,luatexbase.attributes.leetattr)
      then
        if leettable[i.char] then
          i.char = leettable[i.char]
        end
      end
    end
  end
  return head
end
```

10.16 leftsideright

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

```
leftsideright = function(head)
  local factor = 65536/0.99626
  for n in nodetraverseid(GLYPH,head) do
    if (leftsiderightarray[n.char]) then
      shift = nodenew(WHAT,PDF_LITERAL)
      shift2 = nodenew(WHAT,PDF_LITERAL)
      shift.data = "q -1 0 0 1 " .. n.width/factor .." 0 cm"
      shift2.data = "Q 1 0 0 1 " .. n.width/factor .." 0 cm"
      nodeinsertbefore(head,n,shift)
      nodeinsertafter(head,n,shift2)
    end
  end
  return head
end
```

10.17 letterspaceadjust

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

Why the synonym `stealsheep`? Because of a comment of Paul on the `texhax` mailing list: `http://tug.org/pipermail/texhax/2011-October/018374.html`

10.17.1 setup of variables

```
local letterspace_glue   = nodenew(GLUE)
local letterspace_pen    = nodenew(PENALTY)

letterspace_glue.width   = tex.sp"0pt"
letterspace_glue.stretch = tex.sp"0.5pt"
letterspace_pen.penalty  = 10000
```

10.17.2 function implementation

```
letterspaceadjust = function(head)
  for glyph in nodetraverseid(GLYPH, head) do
    if glyph.prev and (glyph.prev.id == GLYPH or glyph.prev.id == DISC or glyph.prev.id == KERN) t
      local g = nodecopy(letterspace_glue)
      nodeinsertbefore(head, glyph, g)
      nodeinsertbefore(head, g, nodecopy(letterspace_pen))
    end
  end
  return head
end
```

10.17.3 textletterspaceadjust

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

```
textletterspaceadjust = function(head)
  for glyph in nodetraverseid(GLYPH, head) do
    if node.has_attribute(glyph,luatexbase.attributes.letterspaceadjustattr) then
      if glyph.prev and (glyph.prev.id == node.id"glyph" or glyph.prev.id == node.id"disc" or gly
        local g = node.copy(letterspace_glue)
        nodeinsertbefore(head, glyph, g)
        nodeinsertbefore(head, g, nodecopy(letterspace_pen))
      end
    end
  end
  luatexbase.remove_from_callback("pre_linebreak_filter","textletterspaceadjust")
  return head
```

```
end
```

10.18 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.

```
matrixize = function(head)
  x = {}
  s = nodenew(DISC)
  for n in nodetraverseid(GLYPH,head) do
    j = n.char
    for m = 0,7 do -- stay ASCII for now
      x[7-m] = nodecopy(n) -- to get the same font etc.

      if (j / (2^(7-m)) < 1) then
        x[7-m].char = 48
      else
        x[7-m].char = 49
        j = j-(2^(7-m))
      end
      nodeinsertbefore(head,n,x[7-m])
      nodeinsertafter(head,x[7-m],nodecopy(s))
    end
    noderemove(head,n)
  end
  return head
end
```

10.19 medievalumlaut

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

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

```
medievalumlaut = function(head)
  local factor = 65536/0.99626
  local org_e_node = nodenew(GLYPH)
  org_e_node.char = 101
  for line in nodetraverseid(0,head) do
    for n in nodetraverseid(GLYPH,line.head) do
      if (n.char == 228 or n.char == 246 or n.char == 252) then
        e_node = nodecopy(org_e_node)
        e_node.font = n.font
        shift = nodenew(WHAT,PDF_LITERAL)
        shift2 = nodenew(WHAT,PDF_LITERAL)
        shift2.data = "Q 1 0 0 1 " .. e_node.width/factor .." 0 cm"
        nodeinsertafter(head,n,e_node)

        nodeinsertbefore(head,e_node,shift)
        nodeinsertafter(head,e_node,shift2)

        x_node = nodenew(KERN)
        x_node.kern = -e_node.width
        nodeinsertafter(head,shift2,x_node)
      end

      if (n.char == 228) then -- ä
        shift.data = "q 0.5 0 0 0.5 " ..
          -n.width/factor*0.85 .." ".. n.height/factor*0.75 .. " cm"
        n.char = 97
      end
      if (n.char == 246) then -- ö
        shift.data = "q 0.5 0 0 0.5 " ..
          -n.width/factor*0.75 .." ".. n.height/factor*0.75 .. " cm"
        n.char = 111
      end
      if (n.char == 252) then -- ü
        shift.data = "q 0.5 0 0 0.5 " ..
          -n.width/factor*0.75 .." ".. n.height/factor*0.75 .. " cm"
        n.char = 117
      end
    end
  end
  return head
end
```

10.20 pancakenize

```
local separator     = string.rep("=", 28)
local texiowrite_nl = texio.write_nl
pancaketext = function()
  texiowrite_nl("Output written on "..tex.jobname..".pdf ("..status.total_pages.." chicken,".." e
  texiowrite_nl(" ")
  texiowrite_nl(separator)
  texiowrite_nl("Soo ... you decided to use \\pancakenize.")
  texiowrite_nl("That means you owe me a pancake!")
  texiowrite_nl(" ")
  texiowrite_nl("(This goes by document, not compilation.)")
  texiowrite_nl(separator.."\n\n")
  texiowrite_nl("Looking forward for my pancake! :)")
  texiowrite_nl("\n\n")
end
```

10.21 randomerror

Not yet implemented, sorry.

10.22 randomfonts

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

```
randomfontslower = 1
randomfontsupper = 0
%
randomfonts = function(head)
  local rfub
  if randomfontsupper > 0 then  -- fixme: this should be done only once, no? Or at every paragrapl
    rfub = randomfontsupper  -- user-specified value
  else
    rfub = font.max()        -- or just take all fonts
  end
  for line in nodetraverseid(Hhead,head) do
    for i in nodetraverseid(GLYPH,line.head) do
      if not(randomfonts_onlytext) or node.has_attribute(i,luatexbase.attributes.randfontsattr) tl
        i.font = math.random(randomfontslower,rfub)
      end
    end
  end
  return head
end
```

10.23 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```
uclcratio = 0.5 -- ratio between uppercase and lower case
randomuclc = function(head)
  for i in nodetraverseid(GLYPH,head) do
    if not(randomuclc_onlytext) or node.has_attribute(i,luatexbase.attributes.randuclcattr) then
      if math.random() < uclcratio then
        i.char = tex.uccode[i.char]
      else
        i.char = tex.lccode[i.char]
      end
    end
  end
  return head
```

```
end
```

10.24 randomchars

```
randomchars = function(head)
  for line in nodetraverseid(Hhead,head) do
    for i in nodetraverseid(GLYPH,line.head) do
      i.char = math.floor(math.random()*512)
    end
  end
  return head
end
```

10.25 randomcolor and rainbowcolor

10.25.1 randomcolor – preliminaries

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

```
randomcolor_grey = false
randomcolor_onlytext = false --switch between local and global colorization
rainbowcolor = false

grey_lower = 0
grey_upper = 900

Rgb_lower = 1
rGb_lower = 1
rgB_lower = 1
Rgb_upper = 254
rGb_upper = 254
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.

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

```
randomcolorstring = function()
  if randomcolor_grey then
    return (0.001*math.random(grey_lower,grey_upper)).." g"
  elseif rainbowcolor then
    if rainind == 1 then -- red
      rainbow_rGb = rainbow_rGb + rainbow_step
      if rainbow_rGb >= 1-rainbow_step then rainind = 2 end
    elseif rainind == 2 then -- yellow
      rainbow_Rgb = rainbow_Rgb - rainbow_step
      if rainbow_Rgb <= rainbow_step then rainind = 3 end
    elseif rainind == 3 then -- green
      rainbow_rgB = rainbow_rgB + rainbow_step
      rainbow_rGb = rainbow_rGb - rainbow_step
      if rainbow_rGb <= rainbow_step then rainind = 4 end
    elseif rainind == 4 then -- blue
      rainbow_Rgb = rainbow_Rgb + rainbow_step
      if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
    else -- purple
      rainbow_rgB = rainbow_rgB - rainbow_step
      if rainbow_rgB <= rainbow_step then rainind = 1 end
    end
    return rainbow_Rgb.." "..rainbow_rGb.." "..rainbow_rgB.." rg"
  else
    Rgb = math.random(Rgb_lower,Rgb_upper)/255
    rGb = math.random(rGb_lower,rGb_upper)/255
    rgB = math.random(rgB_lower,rgB_upper)/255
    return Rgb.." "..rGb.." "..rgB.." ".." rg"
  end
end
```

10.25.2 randomcolor – the function

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

```
randomcolor = function(head)
  for line in nodetraverseid(0,head) do
    for i in nodetraverseid(GLYPH,line.head) do
      if not(randomcolor_onlytext) or
         (node.has_attribute(i,luatexbase.attributes.randcolorattr))
      then
        color_push.data = randomcolorstring()  -- color or grey string
        line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
        nodeinsertafter(line.head,i,nodecopy(color_pop))
      end
    end
  end
  return head
end
```

10.26 randomerror

```
%
```

10.27 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.

```
function relationship()
  sailheight = 12
  mastheight = 4
  hullheight = 5
  relnumber = 402
  shipheight = sailheight + mastheight + hullheight
  tex.print("\\parshape "..(shipheight))
  for i =1,sailheight do
    tex.print(" "..(4.5-i/3.8).."cm "..((i-0.5)/2.5).."cm ")
   end
  for i =1,mastheight do
    tex.print(" "..(3.2).."cm "..(1).."cm ")
  end
  for i =1,hullheight do
    tex.print(" "..((i-1)/2).."cm "..(10-i).."cm ")
  end
  tex.print("\\noindent")
  for i=1,relnumber do
    tex.print("\\ \\char"..math.random(8756,8842))
  end
  tex.print("\\break")
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 horizontial 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`.

```
function cutparagraph(head)
  local parsum = 0
  for n in nodetraverseid(HLIST,head) do
    parsum = parsum + 1
    if parsum > shipheight then
      node.remove(head,n)
    end
  end
  return head
end
```

10.28 rickroll

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

```
%
```

10.29 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.

```
substitutewords_strings = {}

addtosubstitutions = function(input,output)
  substitutewords_strings[#substitutewords_strings + 1] = {}
  substitutewords_strings[#substitutewords_strings][1] = input
  substitutewords_strings[#substitutewords_strings][2] = output
end

substitutewords = function(head)
  for i = 1,#substitutewords_strings do
    head = string.gsub(head,substitutewords_strings[i][1],substitutewords_strings[i][2])
  end
  return head
end
```

10.30 suppressonecharbreak

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

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

```
function suppressonecharbreak(head)
  for i in node.traverse_id(GLUE,head) do
    if ((i.next) and (i.next.next.id == GLUE)) then
        pen = node.copy(suppressonecharbreakpenaltynode)
        node.insert_after(head,i.next,pen)
    end
  end

  return head
end
```

10.31 tabularasa

Removes every glyph from the output and replaces it by empty space. In the end, next to nothing will be visible. Should be extended to also remove rules or just anything visible.

```
tabularasa_onlytext = false

tabularasa = function(head)
  local s = nodenew(KERN)
  for line in nodetraverseid(HLIST,head) do
    for n in nodetraverseid(GLYPH,line.head) do
      if not(tabularasa_onlytext) or node.has_attribute(n,luatexbase.attributes.tabularasaattr) th
        s.kern = n.width
        nodeinsertafter(line.list,n,nodecopy(s))
        line.head = noderemove(line.list,n)
      end
    end
  end
  return head
end
```

10.32 tanjanize

```
tanjanize = function(head)
  local s = nodenew(KERN)
  local m = nodenew(GLYPH,1)
  local use_letter_i = true
  scale = nodenew(WHAT,PDF_LITERAL)
  scale2 = nodenew(WHAT,PDF_LITERAL)
  scale.data  = "0.5 0 0 0.5 0 0 cm"
  scale2.data = "2   0 0 2   0 0 cm"

  for line in nodetraverseid(HLIST,head) do
    for n in nodetraverseid(GLYPH,line.head) do
      mimicount = 0
      tmpwidth  = 0
      while ((n.next.id == GLYPH) or (n.next.id == 11) or (n.next.id == 7) or (n.next.id == 0)) do
find end of a word
        n.next = n.next.next
        mimicount = mimicount + 1
        tmpwidth = tmpwidth + n.width
      end

  mimi = {}  -- constructing the node list.
  mimi[0] = nodenew(GLYPH,1)  -- only a dummy for the loop
  for i = 1,string.len(mimicount) do
   mimi[i] = nodenew(GLYPH,1)
   mimi[i].font = font.current()
   if(use_letter_i) then mimi[i].char = 109 else mimi[i].char = 105 end
   use_letter_i = not(use_letter_i)
   mimi[i-1].next = mimi[i]
  end
--]]

line.head = nodeinsertbefore(line.head,n,nodecopy(scale))
nodeinsertafter(line.head,n,nodecopy(scale2))
   s.kern = (tmpwidth*2-n.width)
   nodeinsertafter(line.head,n,nodecopy(s))
  end
 end
 return head
end
```

10.33 uppercasecolor

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

```
uppercasecolor_onlytext = false

uppercasecolor = function (head)
 for line in nodetraverseid(Hhead,head) do
  for upper in nodetraverseid(GLYPH,line.head) do
   if not(uppercasecolor_onlytext) or node.has_attribute(upper,luatexbase.attributes.uppercase
    if (((upper.char > 64) and (upper.char < 91)) or
      ((upper.char > 57424) and (upper.char < 57451)))  then  -- for small caps! nice
     color_push.data = randomcolorstring()  -- color or grey string
     line.head = nodeinsertbefore(line.head,upper,nodecopy(color_push))
     nodeinsertafter(line.head,upper,nodecopy(color_pop))
    end
   end
  end
 end
 return head
end
```

10.34 upsidedown

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

```
upsidedown = function(head)
 local factor = 65536/0.99626
```

```
  for line in nodetraverseid(Hhead,head) do
    for n in nodetraverseid(GLYPH,line.head) do
      if (upsidedownarray[n.char]) then
        shift = nodenew(WHAT,PDF_LITERAL)
        shift2 = nodenew(WHAT,PDF_LITERAL)
        shift.data = "q 1 0 0 -1 0 " .. n.height/factor .." cm"
        shift2.data = "Q 1 0 0 1 " .. n.width/factor .." 0 cm"
        nodeinsertbefore(head,n,shift)
        nodeinsertafter(head,n,shift2)
      end
    end
  end
  return head
end
```

10.35 colorstretch

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

In fact, two boxes are drawn: The first (left) box shows the badness, i. e. the amount of stretching the spaces between words. Too much space results in 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!

10.35.1 colorstretch – preliminaries

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

```
keeptext = true
colorexpansion = true

colorstretch_coloroffset = 0.5
colorstretch_colorrange = 0.5
chickenize_rule_bad_height = 4/5 -- height and depth of the rules
chickenize_rule_bad_depth = 1/5


colorstretchnumbers = true
drawstretchthreshold = 0.1
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.

```
colorstretch = function (head)
  local f = font.getfont(font.current()).characters
  for line in nodetraverseid(Hhead,head) do
    local rule_bad = nodenew(RULE)

    if colorexpansion then  -- if also the font expansion should be shown
--%% here use first_glyph function!!
      local g = line.head
n = node.first_glyph(line.head.next)
texio.write_nl(line.head.id)
texio.write_nl(line.head.next.id)
texio.write_nl(line.head.next.next.id)
texio.write_nl(n.id)
      while not(g.id == GLYPH) and (g.next) do g = g.next end -- find first glyph on line. If lin
      if (g.id == GLYPH) then                                 -- read width only if g is a glyph!
        exp_factor = g.expansion_factor/10000 --%% neato, luatex now directly gives me this!!
        exp_color = colorstretch_coloroffset + (exp_factor*0.1) .. " g"
texio.write_nl(exp_factor)
        rule_bad.width = 0.5*line.width  -- we need two rules on each line!
      end
    else
      rule_bad.width = line.width  -- only the space expansion should be shown, only one rule
    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.

```
    rule_bad.height = tex.baselineskip.width*chickenize_rule_bad_height -- this should give a bet
    rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth

    local glue_ratio = 0
    if line.glue_order == 0 then
      if line.glue_sign == 1 then
        glue_ratio = colorstretch_colorrange * math.min(line.glue_set,1)
      else
        glue_ratio = -colorstretch_colorrange * math.min(line.glue_set,1)
      end
    end
    color_push.data = colorstretch_coloroffset + glue_ratio .. " g"

```

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

```
-- set up output
    local p = line.head

```

```
-- a rule to immitate kerning all the way back
  local kern_back = nodenew(RULE)
  kern_back.width = -line.width

-- if the text should still be displayed, the color and box nodes are inserted additionally
-- and the head is set to the color node
  if keeptext then
    line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
  else
    node.flush_list(p)
    line.head = nodecopy(color_push)
  end
  nodeinsertafter(line.head,line.head,rule_bad)  -- then the rule
  nodeinsertafter(line.head,line.head.next,nodecopy(color_pop)) -- and then pop!
  tmpnode =  nodeinsertafter(line.head,line.head.next.next,kern_back)

  -- then a rule with the expansion color
  if colorexpansion then  -- if also the stretch/shrink of letters should be shown
    color_push.data = exp_color
    nodeinsertafter(line.head,tmpnode,nodecopy(color_push))
    nodeinsertafter(line.head,tmpnode.next,nodecopy(rule_bad))
    nodeinsertafter(line.head,tmpnode.next.next,nodecopy(color_pop))
  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.

```
  if colorstretchnumbers then
    j = 1
    glue_ratio_output = {}
    for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
      local char = unicode.utf8.char(s)
      glue_ratio_output[j] = nodenew(GLYPH,1)
      glue_ratio_output[j].font = font.current()
      glue_ratio_output[j].char = s
      j = j+1
    end
    if math.abs(glue_ratio) > drawstretchthreshold then
      if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
      else color_push.data = "0 0.99 0 rg" end
    else color_push.data = "0 0 0 rg"
    end

    nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
    for i = 1,math.min(j-1,7) do
      nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])
      end
      nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
    end -- end of stretch number insertion
  end
  return head
end
```

dubstepize

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

```

```

scorpionize

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

```
function scorpionize_color(head)
  color_push.data = ".35 .55 .75 rg"
  nodeinsertafter(head,head,nodecopy(color_push))
  nodeinsertafter(head,node.tail(head),nodecopy(color_pop))
  return head
end
```

10.36 variantjustification

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

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

```
substlist = {}
substlist[1488] = 64289
substlist[1491] = 64290
substlist[1492] = 64291
substlist[1499] = 64292
substlist[1500] = 64293
substlist[1501] = 64294
substlist[1512] = 64295
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").

```
function variantjustification(head)
```

```
  math.randomseed(1)
  for line in nodetraverseid(Hhead,head) do
    if (line.glue_sign == 1 and line.glue_order == 0) then -- exclude the last line!
      substitutions_wide = {} -- we store all "expandable" letters of each line
      for n in nodetraverseid(GLYPH,line.head) do
        if (substlist[n.char]) then
          substitutions_wide[#substitutions_wide+1] = n
        end
      end
      line.glue_set = 0   -- deactivate normal glue expansion
      local width = node.dimensions(line.head)  -- check the new width of the line
      local goal = line.width
      while (width < goal and #substitutions_wide > 0) do
        x = math.random(#substitutions_wide)      -- choose randomly a glyph to be substituted
        oldchar = substitutions_wide[x].char
        substitutions_wide[x].char = substlist[substitutions_wide[x].char] -- substitute by wide
        width = node.dimensions(line.head)             -- check if the line is too wide
        if width > goal then substitutions_wide[x].char = oldchar break end -- substitute back if
        table.remove(substitutions_wide,x)          -- if further substitutions have to be done,
      end
    end
  end
  return head
end
```

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

10.37 zebranize

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

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

10.37.1 zebranize – preliminaries

```
zebracolorarray = {}
zebracolorarray_bg = {}
zebracolorarray[1] = "0.1 g"
zebracolorarray[2] = "0.9 g"
zebracolorarray_bg[1] = "0.9 g"
zebracolorarray_bg[2] = "0.1 g"
```

10.37.2 zebranize – the function

This code has to be revisited, it is ugly.

```
function zebranize(head)
  zebracolor = 1
  for line in nodetraverseid(Hhead,head) do
    if zebracolor == #zebracolorarray then zebracolor = 0 end
    zebracolor = zebracolor + 1
    color_push.data = zebracolorarray[zebracolor]
    line.head =     nodeinsertbefore(line.head,line.head,nodecopy(color_push))
    for n in nodetraverseid(GLYPH,line.head) do
      if n.next then else
        nodeinsertafter(line.head,n,nodecopy(color_pull))
      end
    end

    local rule_zebra = nodenew(RULE)
    rule_zebra.width = line.width
    rule_zebra.height = tex.baselineskip.width*4/5
    rule_zebra.depth = tex.baselineskip.width*1/5

    local kern_back = nodenew(RULE)
    kern_back.width = -line.width

    color_push.data = zebracolorarray_bg[zebracolor]
    line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_pop))
    line.head = nodeinsertbefore(line.head,line.head,nodecopy(color_push))
    nodeinsertafter(line.head,line.head,kern_back)
    nodeinsertafter(line.head,line.head,rule_zebra)
  end
  return (head)
end
```

And that's it! ☺

Well, it's not the whole story so far. I plan to test some drawing using only Lua code, writing directly to the pdf file. This section will grow and get better in parallel to my understandings of what's going on. I.e. it will be very slowly … Nothing here is to be taken as good and/or correct LuaTeXing, and most code is plain ugly. However, it kind of works already ☺

11 Drawing

A *very* first, experimental implementation of a drawing of a chicken. The parameters should be consistent, easy to change and that monster should look more like a cute chicken. However, it is chicken, it is Lua, so it belongs into this package. So far, all numbers and positions are hard coded, this will of course change! The parameters `sloppinessh` and `sloppinessv` give the amount of sloppiness, i. e. how strongly the points are "wiggled" randomly to make the drawings more dynamically. You can set them at any time in the document

```
--
function pdf_print (...)
  for _, str in ipairs({...}) do
    pdf.print(str .. " ")
  end
  pdf.print("\n")
end

function move (p1,p2)
  if (p2) then
    pdf_print(p1,p2,"m")
  else
    pdf_print(p1[1],p1[2],"m")
  end
end

function line(p1,p2)
  if (p2) then
    pdf_print(p1,p2,"l")
  else
    pdf_print(p1[1],p1[2],"l")
  end
end

function curve(p11,p12,p21,p22,p31,p32)
  if (p22) then
    p1,p2,p3 = {p11,p12},{p21,p22},{p31,p32}
  else
    p1,p2,p3 = p11,p12,p21
  end
  pdf_print(p1[1], p1[2],
              p2[1], p2[2],
              p3[1], p3[2], "c")
end

function close ()
  pdf_print("h")
end

function linewidth (w)
  pdf_print(w,"w")
end

function stroke ()
  pdf_print("S")
end
--

function strictcircle(center,radius)
  local left = {center[1] - radius, center[2]}
  local lefttop = {left[1], left[2] + 1.45*radius}
  local leftbot = {left[1], left[2] - 1.45*radius}
  local right = {center[1] + radius, center[2]}
  local righttop = {right[1], right[2] + 1.45*radius}
  local rightbot = {right[1], right[2] - 1.45*radius}

  move (left)
  curve (lefttop, righttop, right)
  curve (rightbot, leftbot, left)
stroke()
end

sloppynessh = 5
sloppynessv = 5

function disturb_point(point)
  return {point[1] + (math.random() - 1/2)*sloppynessh,
          point[2] + (math.random() - 1/2)*sloppynessv}
end

function sloppycircle(center,radius)
  local left = disturb_point({center[1] - radius, center[2]})
  local lefttop = disturb_point({left[1], left[2] + 1.45*radius})
  local leftbot = {lefttop[1], lefttop[2] - 2.9*radius}
  local right = disturb_point({center[1] + radius, center[2]})
  local righttop = disturb_point({right[1], right[2] + 1.45*radius})
  local rightbot = disturb_point({right[1], right[2] - 1.45*radius})

  local right_end = disturb_point(right)

  move (right)
  curve (rightbot, leftbot, left)
  curve (lefttop, righttop, right_end)
  linewidth(math.random()+0.5)
  stroke()
end

function sloppyellipsis(center,radiusx,radiusy)
  local left = disturb_point({center[1] - radiusx, center[2]})
  local lefttop = disturb_point({left[1], left[2] + 1.45*radiusy})
  local leftbot = {lefttop[1], lefttop[2] - 2.9*radiusy}
  local right = disturb_point({center[1] + radiusx, center[2]})
  local righttop = disturb_point({right[1], right[2] + 1.45*radiusy})
  local rightbot = disturb_point({right[1], right[2] - 1.45*radiusy})

  local right_end = disturb_point(right)

  move (right)
  curve (rightbot, leftbot, left)
  curve (lefttop, righttop, right_end)
  linewidth(math.random()+0.5)
  stroke()
end

function sloppyline(start,stop)
  local start_line = disturb_point(start)
  local stop_line = disturb_point(stop)
  start = disturb_point(start)
  stop = disturb_point(stop)
  move(start) curve(start_line,stop_line,stop)
  linewidth(math.random()+0.5)
  stroke()
end
```

12 Known Bugs and Fun Facts

The behaviour of the `\chickenize` macro is under construction and everything it does so far is considered a feature.

babel Using `chickenize` with `babel` leads to a problem with the " (double quote) character, as it is made active: When using `\chickenizesetup` *after* `\begin{document}`, you can *not* use " for strings, but you have to use ' (single quote) instead. No problem really, but take care of this.

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

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

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

13 To Do's

Some things that should be implemented but aren't so far or are very poor at the moment:

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

countglyphs should be extended to count anything the user wants to count

rainbowcolor should be more flexible – the angle of the rainbow should be easily adjustable.

pancakenize should do something funny.

chickenize should differentiate between character and punctuation.

swing swing dancing apes – that will be very hard, actually …

chickenmath chickenization of math mode

14 Literature

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

- 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/

15 Thanks

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