How to read this document.

This is the documentation of the package chickenize. It allows manipulations of any Lua\TeX{} document\footnote{The code is based on pure Lua\TeX{} features, so don’t even try to use it with any other \TeX{} flavour. The package is (partially) tested under plain Lua\TeX{} and (fully) under LuaLa\TeX{}. If you tried using it with Con\TeXt{}, please share your experience, I will gladly try to make it compatible!} exploiting the possibilities offered by the callbacks that influence line breaking (and some other stuff). Most of this package’s content is just for fun and educational use, but there are also some functions that can be useful in a normal production document.

The table on the next page shortly informs you about some of your possibilities and provides links to the (documented) Lua functions. The \TeX{} interface is presented below.

The documentation of this package is far from being well-readable, consistent or even complete. This is caused either by lack of time or priority. If you miss anything that should be documented or if you have suggestions on how to increase the readability of the descriptions, please let me know.

For a better understanding of what’s going on in the code of this package, there is a small tutorial below that explains shortly the most important features used here.

Attention: This package is under development and everything presented here might be subject to incompatible changes. If, by any reason, you decide to use this package for an important document, please make a local copy of the source code and use that. This package will only be considered stable and long-term compatible should it reach version 1.0.

If you have any suggestions or comments, just drop me a mail, I’ll be happy to get any response! The latest source code is hosted on github: https://github.com/alt/chickenize. Feel free to comment or report bugs there, to fork, pull, etc.
**For the Impatient:**

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colorstretch</td>
<td>shows grey boxes that visualise the badness and font expansion line-wise</td>
</tr>
<tr>
<td>letterspaceadjust</td>
<td>improves the greyness by using a small amount of letterspacing</td>
</tr>
<tr>
<td>substitutewords</td>
<td>replaces words by other words (chosen by the user)</td>
</tr>
<tr>
<td>variantjustification</td>
<td>Justification by using glyph variants</td>
</tr>
<tr>
<td>suppressonecharbreak</td>
<td>suppresses linebreaks after single-letter words</td>
</tr>
</tbody>
</table>

### less useful functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boustrophedon</td>
<td>invert every second line in the style of archaic greek texts</td>
</tr>
<tr>
<td>countglyphs</td>
<td>counts the number of glyphs in the whole document</td>
</tr>
<tr>
<td>countwords</td>
<td>counts the number of words in the whole document</td>
</tr>
<tr>
<td>leetspeak</td>
<td>translates the (latin-based) input into 1337 5p34k</td>
</tr>
<tr>
<td>medievalumlaut</td>
<td>changes each umlaut to normal glyph plus “e” above it: åóú</td>
</tr>
<tr>
<td>randomuc1c</td>
<td>alternates randomly between uppercase and lowercase</td>
</tr>
<tr>
<td>rainbowcolor</td>
<td>changes the color of letters slowly according to a rainbow</td>
</tr>
<tr>
<td>randomcolor</td>
<td>prints every letter in a random color</td>
</tr>
<tr>
<td>tabularasa</td>
<td>removes every glyph from the output and leaves an empty document</td>
</tr>
<tr>
<td>uppercasecolor</td>
<td>makes every uppercase letter colored</td>
</tr>
</tbody>
</table>

### complete nonsense

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

---

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

## I User Documentation

1 How It Works ................................................. 6

2 Commands – How You Can Use It
   2.1 \TeX Commands – Document Wide .......................... 6
   2.2 How to Deactivate It ....................................... 9
   2.3 \text Versions .............................................. 9
   2.4 Lua functions ............................................... 9

3 Options – How to Adjust It
   3.1 options for chickenization .................................. 10
   3.2 Options for Game of Chicken ................................. 11

## II Tutorial

4 Lua code ......................................................... 13

5 callbacks ......................................................... 13

6 How to use a callback .......................................... 14

7 Nodes ............................................................... 14

8 Other things ..................................................... 15

## III Implementation

9 \TeX file
   9.1 allownumberincommands ................................... 16
   9.2 drawchicken ................................................. 26
   9.3 drawcov ...................................................... 26
   9.4 drawhorse ................................................... 27

10 \LaTeX package
   10.1 Free Compliments .......................................... 29
   10.2 Definition of User-Level Macros .......................... 29

11 Lua Module
   11.1 chickenize .................................................. 30
   11.2 boustrophedon .............................................. 33
   11.3 bubblesort .................................................. 34

chicken 3
Part I
User Documentation

1 How It Works

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

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

2 Commands – How You Can Use It

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

2.1 \TeX Commands – Document Wide

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

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

\allownumberincommands Normally, you cannot use numbers as part of a control sequence (or, command) name. This makes perfect sense and is good as it is. However, just to raise awareness to this, we provide a command here that changes the 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.\footnote{en.wikipedia.org/wiki/Boustrophedon} 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\footnote{en.wikipedia.org/wiki/Rongorongo} similar style boustrophedon is available with \boustrophedoninverse or \rongorongonize, where subsequent lines are rotated by 180° instead of mirrored.

\footnote{en.wikipedia.org/wiki/Boustrophedon}
\footnote{en.wikipedia.org/wiki/Rongorongo}
\countglyphs  \countwords Counts every printed character (or word, respectively) that appears in anything that is a paragraph. Which is quite everything, in fact. *except* math mode! The total number of glyphs/words will be printed at the end of the log file/console output. For glyphs, also the number of use for every letter is printed separately.

\chickenize Replaces every word of the input with the word “chicken”. Maybe sometime the replacement will be made configurable, but up to now, it’s only chicken. To be a bit less static, about every 10\textsuperscript{th} chicken is uppercase. However, the beginning of a sentence is not recognized automatically.\footnote{If you have a nice implementation idea, I’d love to include this!}

\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 BROOOOAR WOBBBWOBWBWOB BZZZRRRRRRROOOOOOAAAAA ...
... (inspired by \url{http://www.youtube.com/watch?v=ZFQ5Ep07iHk} and \url{http://www.youtube.com/watch?v=nGxpSsbodnw})

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

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

\gameofchicken This is a 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\"\texttt{q}, of course! Additionally, there are \texttt{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.
\anyanize  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 occurrence of word1 will be replaced by word2. You can add replacement pairs by repeated calls to \addtosubstitutions. Take care! This function works with the input stream directly, therefore it does not work on text that is inserted by macros, but it will work on macro names itself! This way, you may use it to change macros (or environments) at will. Bug or feature? I’m not sure right now ...

\suppressonecharbreak  \TeX normally does not suppress a linebreak after words with only one character ("I", "a" etc.) This command suppresses line breaks. It is very similar to the code provided by the 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.

\chicken
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.  

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 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.

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):

\verbatim{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 \verbatim{randomfontslower = 1 randomfontsupper = 0} instead of \verbatim{randomfontslower = 1, randomfontsupper = 0}. Alright?

\footnote{Which is so far not catchable due to missing functionality in luatexbase.}
\footnote{If they don’t have, I did miss that, sorry. Please inform me about such cases.}
\footnote{On a 500 pages text-only \LaTeX{} document the dilation is on the order of 10% with \textrandomcolor{}, but other manipulations can take much more time. However, you are not supposed to make such long documents with chickenize!}
However, \chickenizesetup is a macro on the \TeX side meaning that you can use only \% as comment string. If you use --, all of the argument will be ignored as \TeX does not pass an eol to \directlua. If you don’t understand that, just ignore it and go on as usual.

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

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

3.1 options for chickenization

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

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

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

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

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

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

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

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

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

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

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

chicken 10
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.

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

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

3.2 Options for Game of Chicken

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

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

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

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

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

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

GOCanticellcode = <string> "O" The \LaTeX code for graphical representation of a living anticell.

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

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

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

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

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

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

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

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

chicken 11
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 LuaTeX’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 \latexlua{}. 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 \texttt{tex.} library that offers access to TeX internals. In the simple example above, the function \texttt{tex.print()} inserts its argument into the TeX input stream, so the result of the calculation (10) is printed in the document.

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

\directlua{dofile("filename")}

If you use LuaTeX, you can also use the \texttt{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 \texttt{pre\_linebreak\_filter} and the \texttt{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 \texttt{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 ;)

chicken 13
6 How to use a callback

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

```lua
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. This function has a more extended syntax:

```lua
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:

```lua
luatexbase.remove_from_callback("post_linebreak_filter","a fancy new filter")
```

You have to consult the Lua\TeX manual to see what functionality a callback has when executed, what arguments it expects and what return values have to be given.

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

7 Nodes

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

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

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

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

---

9Since the late 2015 release of \LaTeX, the package has not to be loaded anymore since the functionality is absorbed by the kernel. Plain\TeX users can load the `ltluatex` file which provides the needed functionality.
nodes in a list – e.g. all glyphs in a vertical list that also contains glue, rules etc. This is achieved by calling the function node.traverse_id(GLYPH, head), with the first argument giving the respective id of the nodes.\(^\text{10}\)

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:

```lua
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
```

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

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

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

8 Other things

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

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

The namespace of the chickenize package is not consistent. Please don’t take anything here as an example for good Lua coding, for good \TeX{}ing or even for good Lua\TeX{}ing. 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 ☺

\(^\text{10}\)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 Lua\TeX{} version. We will use this substitute throughout this document.
Part III
Implementation

9 \TeX file

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

For (un)registering, we use the luatexbase \TeX{} 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.

\begin{verbatim}
\directlua{dofile(kpse.find_file("chickenize.lua"))}
\end{verbatim}

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

9.1 allownumberincommands

\begin{verbatim}
\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
}
\end{verbatim}

\begin{verbatim}
\def\BEClerize{
  \chickenize
  \directlua{
    chickenstring[1] = "noise noise"
    chickenstring[2] = "atom noise"
  }
}
\end{verbatim}

\verb|chicken| 16
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[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,"boustrophedon_glyphs")}}
\def\unboustrophedonglyphs{
  \directlua{luatexbase.remove_from_callback("post_linebreak_filter","boustrophedon_glyphs")}}
\def\boustrophedoninverse{
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",boustrophedon_inverse,"boustrophedon_inverse")}}
\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,"detectdoublewords")
    luatexbase.add_to_callback("stop_run",printdoublewords,"printdoublewords")
}
\def\dosomethingfunny{
    %% should execute one of the "funny" commands, but randomly. So every compilation is completely different. Maybe a list of functions could be specified to exclude total nonesense functions. Maybe also on a per-paragraph-basis?
}

c
c
c

\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] = "AAAAAAAH"
    chickenstring[8] = "duhduh duhduh duh"
    chickenstring[9] = "BEEEEEEEW"
    chickenstring[10] = "DDEEEEEEEL"
    chickenstring[12] = "boop"
    chickenstring[13] = "buhdee"
    chickenstring[14] = "bee bee"
    chickenstring[15] = "ZZZZRRRRRRROOOO00000000AAAAA"
    chickenizefraction = 1
  }
  \let\dubstepize\dubstepenize
\def\explainbackslashes{ %% inspired by xkcd #1638
  \tt
  \noindent
  \textbackslash escape character\\
  \textbackslash line end or escaped escape character in tex.print("")\\
  \textbackslash real, real backslash\\
  \textbackslash line end in tex.print("")\\
  \textbackslash elder backslash \\\n  \textbackslash elder backslash to \textbackslash backslash which escapes the screen and enters your brain\!
  \textbackslash elder backslash so real it transcends time and space \!
  \textbackslash elder backslash... the true name of Ba’al, the soul-
  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{

  chicken 19
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\Parencites##1{}
    \gdef\footcite##1{}
    \gdef\Textcite##1{}
    \gdef\textcite##1{}
    \gdef\textcites##1{}
    \gdef\smartcites##1{}
    \gdef\supercite##1{}
    \gdef\Autocite##1{}
  }
}

chicken 20
%% many, many missing ... maybe we need to tackle the underlying mechanism?
\directlua{luatexbase.add_to_callback("pre_linebreak_filter", guttenbergenize_rq, "guttenbergenize")}
\directlua{hammerfirst = true
    luatexbase.add_to_callback("pre_linebreak_filter", hammerweight, "hammerweight")}
\let\hendlnize\chickenize % homage to Hendl/Chicken
\let\unhendlnize\unchickenize % may the soldering strength always be with him
\def\italianizerandwords{
\directlua{luatexbase.add_to_callback("pre_linebreak_filter", italianizerandwords, "italianizerandwords")}
\def\unitalianizerandwords{
\directlua{luatexbase.remove_from_callback("pre_linebreak_filter", "italianizerandwords")}}
\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{luatexbase.remove_from_callback("pre_linebreak_filter", "kernmanipulate")}}
\def\leetspeak{
\directlua{luatexbase.add_to_callback("post_linebreak_filter", 1337, "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
chicken 21
\def\unleftsideright{
  \directlua{luatexbase.remove_from_callback("pre_linebreak\_filter","leftsideright")}}
\def\letterspaceadjust{
  \directlua{luatexbase.add\_to\_callback("pre_linebreak\_filter",letterspaceadjust,"letterspaceadjust")}
  \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}}

\node[draw] (chicken) at (0,0) {chicken 22}
\let\nyanize\rainbowcolor
\let\unyanize\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{ \%
  \directlua{luatexbase.add_to_callback("post_linebreak_filter",randomerror,"randomerror")}
\def\unrandomerror{ \%
  \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\")
  \directlua{luatexbase.add_to_callback("stop_run",missingcharstext,\"charsmissing\")
  \relationship()}
\def\rongorongonize\boustrophedoninverse
\def\unrongorongonize\ unboustrophedoninverse
\def\scorpionize{
  \directlua{luatexbase.add_to_callback("pre_linebreak_filter",scorpionize_color,scorpionize_color")}
\def\unscorpionize{
  \directlua{luatexbase.remove_from_callback("pre_linebreak_filter","scorpionize_color")}}
\def\spankmonkey{ \%
  \directlua{}}
\def\unspankmonkey{ 
chicken 23
\texttt{chicken 24}
Now the setup for the \text{-versions. We utilize Lua\TeX's attributes to mark all nodes that should be manipulated. The macros should be \texttt{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,"textletterspaceadjustactive")
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\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}
\long\def\chickenizesetup#1{%\directlua{#1}}

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{}{\directlua{}{

\texttt{chicken 25}
9.2 drawchicken

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

\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(chickenhead,chickenhead_rad)
  sloppycircle(chickenleg[1][1],chickenleg[1][2]) sloppyline(chickenleg[1][2],chickenleg[1][3])
  sloppylne(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)
}

9.3 drawcov

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

\begin{figure}
\centering
\includegraphics[width=\textwidth]{corona.png}
\caption{Corona virus drawn using the drawcov package.}
\end{figure}
\long\def\drawcov{
\luadraw{90}{
  covbody = {200,50}
  covbody_rad = 50
  covcrown_rad = 5
  crownno = 13
  for i=1,crownno do
    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_rad*math.cos(2*math.pi/crownno*i)}
    crownconnect = {covbody[1]+0.8*covbody_rad*math.sin(2*math.pi/crownno*i),covbody[2]+0.8*covbody_rad*math.cos(2*math.pi/crownno*i)}
    sloppycircle(crownpos,covcrown_rad)
    sloppyline(crownpos,crownconnect)
  end
  covcrown_rad = 8
  sloppycircle(covbody,covcrown_rad)
  sloppycircle(covbody,covbody_rad)
  sloppyline(covbody,covbody)
}
}

9.4 \textbf{drawhorse}

Well … guess what this does.
\long\def\drawhorse{
\luadraw{90}{
  horsebod = {100,-40}
  sloppyellipse(horsebod,50,20)
  horsehead = {20,0}
  sloppyellipse(horsehead,25,15)
  sloppyline({35,-10},{50,-40})
  sloppyline({45,5},{80,-25})
  sloppyline({60,-50},{60,-90})
  sloppyline({70,-50},{70,-90})
  sloppyline({130,-50},{130,-90})
  sloppyline({140,-50},{140,-90})
  sloppyline({150,-40},{160,-60})
}
cocnchicken 27
There’s also a version with a bit more … meat to the bones:

\long\def\drawfathorse{
  \luadraw{90}{
    \def\horsebod{100,-40}
    \def\horsehead{20,0}
    \sloppyline(\horsebod,50,40)
    \sloppyline(\horsehead,25,15)
    \sloppyline({35,-10},{50,-40})
    \sloppyline({45,5},{70,-15})
    \sloppyline({60,-70},{60,-90})
    \sloppyline({70,-70},{70,-90})
    \sloppyline({130,-70},{130,-90})
    \sloppyline({140,-70},{140,-90})
    \sloppyline({150,-40},{160,-60})
    \sloppyline({150,-38},{160,-58})
    \sloppyline({150,-42},{160,-62})
    \sloppyellipse(30,5,5,2) \%
  }
}

% intentionally not documented:
\long\def\drawunicorn{
  \color{pink!90!black}
  \drawhorse
  \luadraw{0}{
    \sloppyline(15,20,15,50)
    \sloppyline(15,50,25,20)
  }
}
\long\def\drawfatunicorn{
  \color{pink!90!black}
  \drawfathorse
  \luadraw{0}{
    \sloppyline(15,20,15,50)
    \sloppyline(15,50,25,20)
  }
}

chicken 28
10 \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}  \\
[2021/01/03 v0.3 chickenize package]  \\
\input{chickenize}

10.1 Free Compliments

\iffalse
\DeclareDocumentCommand\includegraphics{O{}m}{  
\fbox{Chicken}  \% actually, I'd love to draw an MP graph showing a chicken ...
}
\fi

10.2 Definition of User-Level Macros

Nothing done so far, just some minor ideas. If you want to implement some cool things, contact me! :) 
\iffalse
\DeclareDocumentCommand\includegraphics{O{}m}{  \fbox{Chicken}  \% actually, I'd love to draw an MP graph showing a chicken ...
}
\fi

\%
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

11 Lua Module

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

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

\local nodeid = node.id 
\local nodecopy = node.copy 
\local nodenew = node.new 
\local nodetail = node.tail
Now we set up the nodes used for all color things. The nodes are whatsit's 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
```

11.1 chickenize

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

```
chicken_pagenumbers = true

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

chickenize_fraction = 0.5 -- set this to a small value to fool somebody,
-- or to see if your text has been read carefully. This is also a great way to lay easter eggs for
-- chicken_substitutions = 0 -- value to count the substituted chickens. Makes sense for testing your

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)
```

"chicken 30"
while ((i.next.id == GLYPH) or (i.next.id == KERN) or (i.next.id == DISC) or (i.next.id == HLIST)) do -- 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
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 jump
      if (i.char > 64 and i.char < 91) then chickenize_capital = true else chickenize_capital = false end
      head = chickenize_real_stuff(i, head)
    end
  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 == KERN)) then
    chickenize_ignore_word = false
  end
end

return head
end

local separator = string.rep("="

local texiowrite_nl = texio.write_nl

nicetext = function()
  texiowrite_nl("Output written on ".pdf ("..status.total_pages.." chicken,".. eggs.
  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 .. 
  if chickencount then
    texiowrite_nl("There were ".substitutions made.")
  texiowrite_nl(separator)
end

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)

chicken 32
11.2 boustrophedon

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

Linewise rotation:

```javascript
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:

```javascript
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
```

chicken 33
Inverse boustrophedon. At least I think, this is the way Rongorongo is written. However, the top-to-bottom direction has to be inverted, too.

```plaintext
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
```

11.3 bubblesort

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

```plaintext
function bubblesort(head)
    for line in nodetraverse_id(0,head) do
        for glyph in nodetraverse_id(GLYPH,line.head) do
        end
    end
    return head
end
```

11.4 countglyphs

Counts the glyphs in your document. Where "glyph" means every printed character in everything that is a paragraph – formulas do not work! Captions of floats etc. also will not work. However, hyphenations do work and the hyphen sign is counted! And that is the sole reason for this function – every simple script could read the letters in a document, but only after the hyphenation it is possible to count the real number of printed characters – where the hyphen does count.

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

chicken 34
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!

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

To print out the number at the end of the document, the following function is registered in the stop_run callback. This will prevent the normal message from being printed, informing the user about page and memory stats etc. But I guess when counting characters, everything else does not matter at all? ...

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

## 11.5 countwords

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

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

chicken 35
11.6 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
  printdoublewords = function()
  texio.write_nl("finished")
end

11.7 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

11.8 gamofchicken

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

\begin{verbatim}
chicken 36
\end{verbatim}
I also kick in some code to convert the pdf into a gif after the pdf has been finalized and \texttt{Lua\TeX} is about to end. This uses a system call to \texttt{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 \texttt{tex} document.

```latex
833 \textbf{function} \textbf{gameofchicken()} \textbf{end}
834 \hspace{1em} \textbf{GOC\_lifetab} = {} \hspace{1em} \textbf{GOC\_spawntab} = {}
835 \hspace{1em} \textbf{GOC\_antilifetab} = {} \hspace{1em} \textbf{GOC\_antispawntab} = {}
836
837 \hspace{1em} -- \textit{translate the rules into an easily-manageable table}
838 \hspace{1em} \textbf{for} i=1,\#\textbf{GOC\_rule\_live} \textbf{do} \hspace{.5em} \textbf{GOC\_lifetab}[\textbf{GOC\_rule\_live}[i]] = \textbf{true} \hspace{1em} \textbf{end}
839 \hspace{1em} \textbf{for} i=1,\#\textbf{GOC\_rule\_spawn} \textbf{do} \hspace{.5em} \textbf{GOC\_spawntab}[\textbf{GOC\_rule\_spawn}[i]] = \textbf{true} \hspace{1em} \textbf{end}
840 \hspace{1em} \textbf{for} i=1,\#\textbf{GOC\_rule\_antilive} \textbf{do} \hspace{.5em} \textbf{GOC\_antilifetab}[\textbf{GOC\_rule\_antilive}[i]] = \textbf{true} \hspace{1em} \textbf{end}
841 \hspace{1em} \textbf{for} i=1,\#\textbf{GOC\_rule\_antispawn} \textbf{do} \hspace{.5em} \textbf{GOC\_antispawntab}[\textbf{GOC\_rule\_antispawn}[i]] = \textbf{true} \hspace{1em} \textbf{end}
842
843 \hspace{1em} \textit{Initialize the arrays for cells and anticells with zeros.}
844 \hspace{1em} \textbf{local} \textbf{life} = {} \hspace{1em} \textbf{local} \textbf{antilife} = {} \hspace{1em} \textbf{local} \textbf{newlife} = {} \hspace{1em} \textbf{local} \textbf{newantilife} = {}
845 \hspace{1em} \textbf{for} i = 0, \textbf{GOC\_x} \textbf{do} \textbf{life}[i] = {}; \textbf{newlife}[i] = {} \textbf{end}
846 \hspace{1em} \textbf{for} i = 0, \textbf{GOC\_x} \textbf{do} \textbf{antilife}[i] = {}; \textbf{newantilife}[i] = {} \textbf{end}
847
848 \hspace{1em} \textit{These are the functions doing the actual work, checking the neighbors and applying the rules defined above.}
849 \hspace{1em} \textbf{function} \textbf{applyrules\_life}(\textbf{neighbors}, \textbf{lifeij}, \textbf{antineighbors}, \textbf{antilifeij}) \textbf{end}
850 \hspace{1em} \textbf{if} \textbf{GOC\_spawntab}[\textbf{neighbors}] \textbf{then} \textbf{myret} = 1 \textbf{else} 1 \textbf{end}
851 \hspace{2em} \textbf{if} \textbf{GOC\_lifetab}[\textbf{neighbors}] \textbf{and} (\textbf{lifeij} == 1) \textbf{then} \textbf{myret} = 1 \textbf{else} \textbf{myret} = 0 \textbf{end}
852 \hspace{2em} \textbf{if} \textbf{antineighbors} > 1 \textbf{then} \textbf{myret} = 0 \textbf{else} \textbf{myret} = 0 \textbf{end}
853 \hspace{1em} \textbf{return} \textbf{myret}
854 \hspace{1em} \textbf{end}
855 \hspace{1em} \textbf{function} \textbf{applyrules\_antilife}(\textbf{neighbors}, \textbf{lifeij}, \textbf{antineighbors}, \textbf{antilifeij}) \textbf{end}
856 \hspace{1em} \textbf{if} \textbf{(antineighbors} == 3) \textbf{then} \textbf{myret} = 1 \textbf{else} \textbf{myret} = 1 \textbf{end}
857 \hspace{2em} \textbf{if} \textbf{(antineighbors} > 1) \textbf{and} (\textbf{antineighbors} < 4) \textbf{and} (\textbf{lifeij} == 1) \textbf{then} \textbf{myret} = 1 \textbf{else} \textbf{myret} = 0 \textbf{end}
858 \hspace{2em} \textbf{if} \textbf{neighbors} > 1 \textbf{then} \textbf{myret} = 0 \textbf{else} \textbf{myret} = 0 \textbf{end}
859 \hspace{1em} \textbf{return} \textbf{myret}
860 \hspace{1em} \textbf{end}
861
862 \hspace{1em} \textit{Preparing the initial state with a default pattern:}
863 \hspace{1em} \textbf{-- prepare some special patterns as starter}
864 \hspace{1em} \textbf{life}[53][26] = 1 \hspace{1em} \textbf{life}[53][25] = 1 \hspace{1em} \textbf{life}[54][25] = 1 \hspace{1em} \textbf{life}[55][25] = 1 \hspace{1em} \textbf{life}[54][24] = 1
865
866 \hspace{1em} \textit{And the main loop running from here:}
867 \hspace{1em} \textbf{print("start");}
868 \hspace{1em} \textbf{for} i = 1, \textbf{GOC\_x} \textbf{do}
869 \hspace{2em} \textbf{for} j = 1, \textbf{GOC\_y} \textbf{do}
870 \hspace{3em} \textbf{if} (\textbf{life}[i][j]==1) \textbf{then} \textbf{texio\_write("X")} \textbf{else} \textbf{if} (\textbf{antilife}[i][j]==1) \textbf{then} \textbf{texio\_write("O")} \textbf{else}
871 \hspace{4em} \textbf{texio\_write("_")}
872 \hspace{3em} \textbf{end}
873 \hspace{2em} \textbf{end}
874 \hspace{1em} \textbf{end}
```

\[\textit{chicken 37}\]
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[i+1][j+1])
      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 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[i+1][j+1])
        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
  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("O") else texio.write("_") end
      end
      if GOC_pdf then
        if (life[i][j]==1) then tex.print("\placeat(\.(i/10)..\.(j/10).\){{\hfill\hfillGOCcellcode.\hfill\hfill}}") end
        if (antilife[i][j]==1) then tex.print("\placeat(\.(i/10)..\.(j/10).\){{\hfill\hfillGOCanticellcode.\hfill\hfill}}") end
      end
    end
  end
  tex.print(\newpage)
  os.sleep(GOCsleep)
end

chicken 38
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.

```lua
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
```

### 11.9 guttenbergenize

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

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

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

#### 11.9.1 guttenbergenize – preliminaries

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

```lua
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,
}
```

#### 11.9.2 guttenbergenize – the function

```lua
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
```

---

Thank you to Jasper for bringing me to this idea!
11.10 hammertime

This is a completely useless function. It just prints STOP! – HAMMERTIME at the beginning 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 Lua\TeX{} mailing list.\footnote{http://tug.org/pipermail/luatex/2011-November/003355.html}

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

11.11 italianize

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

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

chicken 40
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 math.random() > 0.2 then
            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

11.12 italianizerandwords

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

italianizerandwords = function(head)
words = {}

chicken 41
wordnumber = 0
-- head.next.next is the very first word. However, let's try to get the first word after the first space correct.
for n in nodetraverseid(GLUE, head) do -- let's try to count words by their separators
    wordnumber = wordnumber + 1
    if n.next then
        words[wordnumber] = {}
        words[wordnumber][1] = node.copy(n.next)
    end
    glyphnumber = 1
    myglyph = n.next
    while myglyph.next do
        node.tail(words[wordnumber][1]).next = node.copy(myglyph.next)
        myglyph = myglyph.next
    end
end
print(#words)
if #words > 0 then
    print("lengs is: ")
end
print(#words[#words])
end
--myinsertnode = head.next.next -- first letter
--node.tail(words[1][1]).next = myinsertnode.next
--myinsertnode.next = words[1][1]
return head

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.")
end

chicken 42
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 count one word too much.
else
wordlist[wordnumber] = wordlist[wordnumber] + 1 -- the current word got 1 glyph longer
end
end
return head
end

11.13 itsame

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

itsame = function()
local mr = function(a,b) rectangle({a*10,b*-10},10,10) end
local 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)
for i = 2,4 do mr(4,i,8) end
for i = 2,4 do mr(5,i,5) end
for i = 2,4 do mr(6,i,6) end
for i = 3,11 do mr(10,i,11) end
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

chicken 43
1093 for i = 5,8 do mr(i,10) end
1094 mr(8,9) mr(4,11) mr(6,11) mr(7,11) mr(9,11)
1095 for i = 4,9 do mr(i,12) end
1096 for i = 3,10 do mr(i,13) end
1097 for i = 3,5 do mr(i,14) end
1098 for i = 7,10 do mr(i,14) end
1099 end

11.14 kernmanipulate

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

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

chickenkernamount = 0
chickeninvertkerning = false

1111 function kernmanipulate (head)
1112 if chickeninvertkerning then -- invert the kerning
1113 for n in nodetraverseid(11,head) do
1114 n.kern = -n.kern
1115 end
1116 else -- if not, set it to the given value
1117 for n in nodetraverseid(11,head) do
1118 n.kern = chickenkernamount
1119 end
1120 end
1121 return head
1122 end

11.15 leetspeak

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

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

```lua
function leet(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
```

### 11.16 leftsideright

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

```lua
function leftsideright(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
```

### 11.17 letterspaceadjust

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

chicken 45

11.17.1 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
```

11.17.2 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) then
      local g = nodecopy(letterspace_glue)
      nodeinsertbefore(head, glyph, g)
      nodeinsertbefore(head, g, nodecopy(letterspace_pen))
    end
  end
  return head
end
```

11.17.3 textletterspaceadjust

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

```
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 glyph.prev.id == KERN) then
        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
```

11.18 matrixize

Substitutes every glyph by a representation of its ASCII value. Might be extended to cover the entire unicode range, but so far only 8bit is supported. The code is quite straight-forward and works OK. The line ends are

```
chicken 46
```
not necessarily adjusted correctly. However, with microtype, i.e. font expansion, everything looks fine.

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

11.19 medievalumlaut

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

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

```plaintext
1209 medievalumlaut = function(head)
1210 local factor = 65536/0.99626
1211 local org_e_node = nodenew(GLYPH)
1212 org_e_node.char = 101
1213 for line in nodetraverseid(0,head) do
1214 for n in nodetraverseid(GLYPH,line.head) do
1215 if (n.char == 228 or n.char == 246 or n.char == 252) then
1216 e_node = nodecopy(org_e_node)
1217 e_node.font = n.font
1218 shift = nodenew(WHAT,PDF_LITERAL)
1219 shift2 = nodenew(WHAT,PDF_LITERAL)
1220 shift2.data = "Q 1 0 0 1 " .. e_node.width/factor .." 0 cm"
1221 nodeinsertafter(head,n,e_node)
1222 nodeinsertbefore(head,e_node,shift)
1223 end
1224 end
1225 return head
1226 end

chicken 47
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
return head

11.20 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,".. egg
	exiowrite_nl(" ")
	exiowrite_nl(separator)
	exiowrite_nl("Soo ... you decided to use \pancakenize.")	exiowrite_nl("That means you owe me a pancake!")	exiowrite_nl(" ")	exiowrite_nl("This goes by document, not compilation.")	exiowrite_nl(separator.."\n\n")	exiowrite_nl("Looking forward for my pancake! :)"
	exiowrite_nl("\n\n")
end

11.21 randomerror

Not yet implemented, sorry.

\frac$
11.22 randomfonts

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

```latex
11.22 randomfonts

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

\begin{verbatim}
randomfontslower = 1
randomfontsupper = 0

randomfonts = function(head)
    local rfub
    if randomfontsupper > 0 then -- fixme: this should be done only once, no? Or at every paragraph?
        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) then
                i.font = math.random(randomfontslower,rfub)
            end
        end
    end
    return head
end
\end{verbatim}

11.23 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```latex
11.23 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```latex
\begin{verbatim}
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.randuclcatr) 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
\end{verbatim}

11.24 randomchars

```latex
11.24 randomchars

```latex
\begin{verbatim}
randomchars = function(head)
    for line in nodetraverseid(Hhead,head) do
        for i in nodetraverseid(GLYPH,line.head) do
            if not(randomchars_onlytext) or node.has_attribute(i,luatexbase.attributes.randcharsattr) then
                i.font = math.random(randomcharslower,rfub)
            end
        end
    end
    return head
end
\end{verbatim}

chicken 49
i.char = math.floor(math.random() * 512)
end
end
return head
end

11.25 randomcolor and rainbowcolor

11.25.1 randomcolor – preliminaries

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

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
    elseif rainind == 2 then -- yellow
        rainbow_Rgb = rainbow_Rgb - rainbow_step
    elseif rainind == 3 then -- green
        rainbow_rgB = rainbow_rgB - rainbow_step
    end
    if rainbow_rGb <= rainbow_step then rainind = 2 end
    if rainbow_Rgb <= rainbow_step then rainind = 3 end
    if rainbow_rgB <= rainbow_step then rainind = 4 end
end

chicken 50
1335     if rainbow_rGb <= rainbow_step then rainind = 4 end
1336   elseif rainind == 4 then -- blue
1337     rainbow_Rgb = rainbow_Rgb + rainbow_step
1338     if rainbow_Rgb >= 1-rainbow_step then rainind = 5 end
1339   else -- purple
1340     rainbow_rgB = rainbow_rgB - rainbow_step
1341     if rainbow_rgB <= rainbow_step then rainind = 1 end
1342   end
1343   return rainbow_Rgb.."..rainbow_rGb.."..rainbow_rgB.." rg"
1344 else
1345   Rgb = math.random(Rgb_lower,Rgb_upper)/255
1346   rGb = math.random(rGb_lower,rGb_upper)/255
1347   rgB = math.random(rgB_lower,rgB_upper)/255
1348   return Rgb.."..rGb.."..rgB..".. rg"
1349 end
1350 end

11.25.2 randomcolor – the function

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

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

11.26 relationship

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

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

chicken 51
--% check if there's a problem with any character in the current font
f = font.getfont(font.current())
fullfont = 1
for i = 8756,8842 do
  if not(f.characters[i]) then texio.write_nl((i)..' not available') fullfont = 0 end
end
--% store the result of the check for later, then go on to construct the ship:
shipheight = sailheight + mastheight + hullheight
tex.print("\parshape ".(shipheight)) --% prepare the paragraph shape ...
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") --% ... up to here, then insert relations
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 horizontal space. So we cannot make sure the same number of lines for each version. To catch this, we typeset more lines and just remove excess lines with a simple function in our beloved post_linebreak_filter.

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

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

function missingcharstext()
if (fullfont == 0) then
  local separator = string.rep("=", 28)
  local texiowrite_nl = texio.write_nl
  texiowrite_nl("Output written on ..tex.jobname..\.pdf (..status.total_pages.." chicken,".. egg

chicken 52
\relationship needs special characters (unicode points 8756 to 8842)
Your font does not support all of them!
consider using another one, e.g. the XITS font supplied with TeXlive.
end

11.27 \textbf{rickroll}

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

\% 

11.28 \textbf{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 substituteword which is registered in the process_input_buffer callback. Once the substitution list is built, the rest is very simple: We just use gsub to substitute, do this for every item in the list, and that's it.

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

11.29 \textbf{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

chicken 53
it, but the simplest. The penalty is not created newly each time, but copied – no significant speed gain, however.

```literate
1433 suppressonecharbreakpenaltynode = node.new(PENALTY)
1434 suppressonecharbreakpenaltynode.penalty = 10000
1435 function suppressonecharbreak(head)
1436   for i in node.traverse_id(GLUE,head) do
1437     if ((i.next) and (i.next.next.id == GLUE)) then
1438       pen = node.copy(suppressonecharbreakpenaltynode)
1439       node.insert_after(head,i.next,pen)
1440     end
1441   end
1442 return head
1443 end
```

### 11.30 tabularasa

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

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

### 11.31 tanjanize

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

*chicken* 54
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

11.32 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.uppercasecolorattr) then
        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

chicken 55
11.33 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

11.34 colorstretch

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

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

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

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

11.34.1 colorstretch – preliminaries

Two booleans, keepertext, 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 line is empty, no glyph:
            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 better
        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)
            end
        end

chicken 57
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 imitate 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,tmpnode.next.next,nodecopy(color_pop))

-- 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

chicken 58
1613     j = j+1
1614   end
1615   if math.abs(glue_ratio) > drawstretchthreshold then
1616     if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
1617     else color_push.data = "0 0.99 0 rg" end
1618   else color_push.data = "0 0 0 rg"
1619   end
1620   nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_push))
1621   for i = 1,math.min(j-1,7) do
1622     nodeinsertafter(line.head,node.tail(line.head),glue_ratio_output[i])
1623   end
1624   nodeinsertafter(line.head,node.tail(line.head),nodecopy(color_pop))
1625   end -- end of stretch number insertion
1626 end
1627
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

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

11.35 variantjustification

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

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

1637 substlist = {}
1638 substlist[1488] = 64289
1639 substlist[1491] = 64290
1640 substlist[1492] = 64291
1641 substlist[1499] = 64292
In the function, we need reproducible randomization so every compilation of the same document looks the same. Else this would make contracts invalid.

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

11.36 zebranize

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

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

\textit{chicken} / 60
11.36.1 zebranize – preliminaries

```plaintext
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"
```

11.36.2 zebranize – the function

This code has to be revisited, it is ugly.

```plaintext
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
    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 ☺

12 Drawing

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

```
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],
```

chicken 62
By setting `drawwidth` to something different than 1 you can adjust the thickness of the strokes. Any stroke done with the `sloppy` functions will by varied between 0.5 `drawwidth` and 1.5 `drawwidth`.

```plaintext
drawwidth = 1
```

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

```plaintext
function stroke ()
    pdf_print("S")
end
```

```plaintext
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
```

```plaintext
sloppynessh = 5
sloppynessv = 5
```

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

```plaintext
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}
end
```

*chicken 63*
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(drawwidth*(math.random()+0.5))
stroke()

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(drawwidth*(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(drawwidth*(math.random()+0.5))
stroke()
end

chicken 64
13 Known Bugs and Fun Facts

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

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

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

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

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

14 To Do’s

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

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

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

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

**pancakenize** should do something funny.

**chickenize** should differentiate between character and punctuation.

**swing** swing dancing apes – that will be very hard, actually ...

**chickenmath** chickenization of math mode

15 Literature

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

- LuaTeX documentation – the manual and links to presentations and talks: [http://www.luatex.org/documentation.html](http://www.luatex.org/documentation.html)
- The Lua manual, for Lua 5.1: [http://www.lua.org/manual/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/](http://www.lua.org/pil/)

*chicken* 65
16 Thanks

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

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