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

Paul Isambert

CHICKENIZE

v0.1
Arno Trautmann
arno.trautmann@gmx.de

This is the documentation of the package `chickenize`. It allows manipulations of any LuaTEX document[1] exploiting the possibilities offered by the callbacks that influence line breaking (and some other stuff). Most of this package's content is just for fun and educational use, but there are also some functions that can be useful in a normal 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 not be considered stable until it reaches at least v0.5.

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

[1]The code is based on pure LuaTEX features, so don't even try to use it with any other TEX flavour. The package is tested under plain LuaTEX and LuaLATEX. If you tried using it with ConTEXt, please share your experience, I will gladly try to make it compatible!

For the Impatient:

A small and incomplete overview of the functionalities offered by this package. I try to keep this list as complete as possible. Of course, the label "complete nonsense" depends on what you are doing …

	maybe useful functions
colorstretch	shows grey boxes that visualise the badness and font expansion of each line
letterspaceadjust	improves the greyness by using a small amount of letterspacing
substitutewords	replaces words by other words (user-controlled!)
	less useful functions
countglyphs	counts the number of glyphs in the whole document
leetspeak	translates the (latin-based) input into 1337 5p34k
randomuclc	alternates randomly between uppercase and lowercase
rainbowcolor	changes the color of letters slowly according to a rainbow
randomcolor	prints every letter in a random color
tabularasa	removes every glyph from the output and leaves an empty document
uppercasecolor	makes every uppercase letter colored
	complete nonsense
chickenize	replaces every word with "chicken" (or user-adjustable words)
guttenbergenize	deletes every quote and footnotes
hammertime	U can't touch this!
kernmanipulate	manipulates the kerning (tbi)
matrixize	replaces every glyph by its ASCII value in binary code
randomerror	just throws random (La)TEX errors at random times
randomfonts	changes the font randomly between every letter
randomchars	randomizes the (letters of the) whole input

Contents

Part I
User Documentation

1 How It Works

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

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

2 Commands – How You Can Use It

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

`\countglyphs` Counts every printed character that appeared in anything that is a paragraph. Which is quite everything, in fact, *exept* math mode! The total number will be printed at the end of the log file/console output.

`\chickenize` Replaces every word of the input with the word "chicken". Maybe sometime the replacement will be made configurable, but up to now, it's only chicken. To be a bit less static, about every 10th chicken is uppercase. However, the beginning of a sentence is not recognized automatically.[2]

`\dubstepize` wub wub wub wub wub BROOOOOAR WOBBBWOBBWOBB BZZZRRRRRRROOOOOOAAAAA … (inspired by `http://www.youtube.com/watch?v=ZFQ5EpO7iHk` and `http://www.youtube.com/watch?v=nGxpSsbodnw`)

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

`\hammertime` STOP! —— Hammertime!

`\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.

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

`\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.

`\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.

`\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.

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

`\nyanize` A synonym for `rainbowcolor`.

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

`\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.

`\substitutewords` You have to specify pairs of words by using `\addtosubstitutions{word1}{word2}`. Then call `\substitutewords` (or the other way round, doesn't matter) and each occurance of `word1` will be replaced by `word2`. You can add replacement pairs by repeated calls to `\addtosubstitutions`. Take care! This function warks with the input 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 …

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.[3]

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.

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

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[4] 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.[5]

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

2.4 Lua functions

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

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

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

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

3 Options – How to Adjust It

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

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.

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

[5] 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`!

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

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

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

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

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

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

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

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

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

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

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

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

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

Part II
Tutorial

I thought it might be helpful to add a small tutorial to this package as it is mainly written with instructional purposes in mind. However, the following is *not* intended as a comprehensive guide to LuaTEXİt's just to get an idea how things work here. For a deeper understanding of LuaTEX you should consult both the LuaTEX manual and some introduction into Lua proper like "Programming in Lua". (See the section Literature at the end of the manual.)

4 Lua code

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

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

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

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

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

If you use LuaLATEX, you can also use the `luacode` environment from the eponymous package.

5 callbacks

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

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

5.1 How to use a callback

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

```
function my_new_filter(head)
  return head
end

callback.register("post_linebreak_filter",my_new_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 package `luatexbase` by Manuel Pégourié-Gonnard and Élie Roux that offers the function `luatexbase.add_to_callback` which has a somewhat extended syntax:

```
luatexbase.add_to_callback("post_linebreak_filter",my_new_filter,"a fancy new filter")
```

The third argument is a name you can (have to) give to your function in the callback. That is necessary because the package also allows for removing functions from callbacks, and then you need a unique identifier for the function:

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

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

Everything I have written here is not the complete truth – please consult the LuaTEX manual and the `luatexbase` documentation for details!

6 Nodes

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

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

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

A more convenient way to adress each node of a list is the function `node.traverse(head)` which takes as first argument the first node of the list. However, often one wants to adress only a certain type of nodes in a list – e. g. all glyphs in a vertical list that also contains glue, rules etc. This is achieved by calling the function `node.traverse_id(37,head)`, with the first argument giving the respective id of the nodes.

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

```
function remove_e(head)
```

```
  for n in node.traverse_id(37,head) do
    if n.char == 101 then
      node.remove(head,n)
    end
  end
  return head
end

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

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

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

7 Other things

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

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

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

Part III

Implementation

8 TEX file

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

For (un)registering, we use the `luatexbase` package. 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`.

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

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

\def\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{glyphnumber = 0
             luatexbase.add_to_callback("post_linebreak_filter",countglyphs,"countglyphs")
             luatexbase.add_to_callback("stop_run",printglyphnumber,"printglyphnumber")
  }
}

\def\dosomethingfunny{
    %% should execute one of the "funny" commands, but randomly. So every compilation is complete
  }

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

    chickenizefraction = 1
  }
}
\let\dubstepize\dubstepenize

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
\newluatexattribute\leetattr
\newluatexattribute\randcolorattr
\newluatexattribute\randfontsattr
\newluatexattribute\randuclcattr
\newluatexattribute\tabularasaattr
\newluatexattribute\uppercasecolorattr

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

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

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

The following is the very first try of implementing a small drawing language in Lua. It draws a beautiful chicken.

```
\long\def\luadraw#1#2{%
  \vbox to #1bp{%
    \vfil
    \luatexlatelua{pdf_print("q") #2 pdf_print("Q")}%
  }%
}
\long\def\drawchicken{
\luadraw{90}{
kopf = {200,50} % Kopfmitte
kopf_rad = 20

d = {215,35} % Halsansatz
e = {230,10} %

korper = {260,-10}
korper_rad = 40

bein11 = {260,-50}
bein12 = {250,-70}
bein13 = {235,-70}

bein21 = {270,-50}
bein22 = {260,-75}
bein23 = {245,-75}

schnabel_oben = {185,55}
schnabel_vorne = {165,45}
schnabel_unten = {185,35}

flugel_vorne = {260,-10}
flugel_unten = {280,-40}
flugel_hinten = {275,-15}

sloppycircle(kopf,kopf_rad)
sloppyline(d,e)
sloppycircle(korper,korper_rad)
sloppyline(bein11,bein12) sloppyline(bein12,bein13)
sloppyline(bein21,bein22) sloppyline(bein22,bein23)
sloppyline(schnabel_vorne,schnabel_oben) sloppyline(schnabel_vorne,schnabel_unten)
sloppyline(flugel_vorne,flugel_unten) sloppyline(flugel_hinten,flugel_unten)
}
}
```

9 LaTeX package

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

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

```
\ProvidesPackage{chickenize}%
  [2012/05/20 v0.1 chickenize package]
\input{chickenize}
```

9.1 Definition of User-Level Macros

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

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

10 Lua Module

This file contains all the necessary functions and is the actual work horse of this package. The functions are sorted strictly 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 nodenew = node.new
local nodecopy = node.copy
local nodeinsertbefore = node.insert_before
local nodeinsertafter = node.insert_after
local noderemove = node.remove
local nodeid = node.id
local nodetraverseid = node.traverse_id
local nodeslide = node.slide

Hhead = nodeid("hhead")
RULE = nodeid("rule")
GLUE = nodeid("glue")
WHAT = nodeid("whatsit")
COL = node.subtype("pdf_colorstack")
GLYPH = nodeid("glyph")
```

Now we set up the nodes used for all color things. The nodes are whatsits of subtype `pdf_colorstack`.

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

10.1 chickenize

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

```
chicken_pagenumbers = true

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

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

local tbl = font.getfont(font.current())
local space = tbl.parameters.space
local shrink = tbl.parameters.space_shrink
local stretch = tbl.parameters.space_stretch
local match = unicode.utf8.match
chickenize_ignore_word = false
```

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

```
chickenize_real_stuff = function(i,head)
    while ((i.next.id == 37) or (i.next.id == 11) or (i.next.id == 7) or (i.next.id == 0)) do  --
      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-docum
-- 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(37,1)  -- only a dummy for the loop
    for i = 1,string.len(chickenstring_tmp) do
      chicken[i] = nodenew(37,1)
      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(10)
        chicken[j].spec = nodenew(47)
        chicken[j].spec.width = space
        chicken[j].spec.shrink = shrink
        chicken[j].spec.stretch = stretch
      end
      j = j+1
    end

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

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

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

  return head
end

chickenize = function(head)
  for i in nodetraverseid(37,head) do  --find start of a word
    if (chickenize_ignore_word == false) then  -- normal case: at the beginning of a word, we jum
      if (i.char > 64 and i.char < 91) then chickenize_capital = true else chickenize_capital = f
      head = chickenize_real_stuff(i,head)
    end

-- At the end of the word, the ignoring is reset. New chance for everyone.
    if not((i.next.id == 37) or (i.next.id == 7) or (i.next.id == 22) or (i.next.id == 11)) then
      chickenize_ignore_word = false
    end

-- And the 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
  end
  return head
end

```

A small additional feature: Some nice text to cheer up the user. Mainly to show that and how we can access the `stop_run` callback. (see above)

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

10.2 countglyphs

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

This function will be extended to allow counting of whatever you want.

```
countglyphs = function(head)
  for line in nodetraverseid(0,head) do
    for glyph in nodetraverseid(37,line.head) do
      glyphnumber = glyphnumber + 1
    end
  end
  return head
end
```

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

```
printglyphnumber = function()
  texiowrite_nl("Number of glyphs in this document: "..glyphnumber)
end
```

10.3 guttenbergenize

A function in honor of the German politician Guttenberg.[6] 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.

10.3.1 guttenbergenize – preliminaries

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

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

10.3.2 guttenbergenize – the function

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

10.4 hammertime

This is a completely useless function. It just prints STOP! – HAMMERTIME at the beginnig of the first paragraph after `\hammertime`, and "U can't touch this" for every following one. As the function writes to the terminal, you have to be sure that your terminal is line-buffered and not block-buffered. Compare the explanation by Taco on the LuaTEX mailing list.[7]

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

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

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

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

10.5 itsame

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

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

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

color = "1 0 0"
```

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

10.6 kernmanipulate

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

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

```
chickenkernamount = 0
chickeninvertkerning = false

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

10.7 leetspeak

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

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

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

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

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

10.8 letterspaceadjust

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

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

10.8.1 setup of variables

```
local letterspace_glue = nodenew(nodeid"glue")
local letterspace_spec = nodenew(nodeid"glue_spec")
local letterspace_pen = nodenew(nodeid"penalty")

letterspace_spec.width   = tex.sp"0pt"
letterspace_spec.stretch = tex.sp"2pt"
letterspace_glue.spec    = letterspace_spec
letterspace_pen.penalty  = 10000
```

10.8.2 function implementation

```
letterspaceadjust = function(head)
```

```
  for glyph in nodetraverseid(nodeid"glyph", head) do
    if glyph.prev and (glyph.prev.id == nodeid"glyph" or glyph.prev.id == nodeid"disc") then
      local g = nodecopy(letterspace_glue)
      nodeinsertbefore(head, glyph, g)
      nodeinsertbefore(head, g, nodecopy(letterspace_pen))
    end
  end
  return head
end
```

10.8.3 restricted letterspaceadjust

To let the user choose the beginning and end of letterspaceadjust, the macros `\startlsa` and `\stoplsa` are provided. The rest ist similar to the `\text...` macros, with one big difference here: We try to use the function to unregister itself. This way, no explicit call to the function is needed anymore. FIXME: not working so far due to missing feature of luatexbase ...

```
letterspaceadjust_restricted = function(head)
  for glyph in nodetraverseid(nodeid"glyph", head) do
    if glyph.prev and (glyph.prev.id == nodeid"glyph" or glyph.prev.id == nodeid"disc") then
      local g = nodecopy(letterspace_glue)
      nodeinsertbefore(head, glyph, g)
      nodeinsertbefore(head, g, nodecopy(letterspace_pen))
    end
  end
  luatexbase.remove_from_callback()
  return head
end
```

10.9 matrixize

Substitutes every glyph by a representation of its ASCII value. Migth be extended to cover the entire unicode range, but so far only 8bit is supported. The code is quite straight-forward and works OK. The line ends are not necessarily adjusted correctly. However, with microtype, i. e. font expansion, everything looks fine.

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

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

10.10 pancakenize

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

10.11 randomerror

10.12 randomfonts

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

```
randomfontslower = 1
randomfontsupper = 0
%
randomfonts = function(head)
  local rfub
  if randomfontsupper > 0 then  -- fixme: this should be done only once, no? Or at every paragrap
    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) t
        i.font = math.random(randomfontslower,rfub)
      end
    end
  end
  return head
end
```

10.13 randomuclc

Traverses the input list and changes lowercase/uppercase codes.

```
uclcratio = 0.5 -- ratio between uppercase and lower case
randomuclc = function(head)
  for i in nodetraverseid(37,head) do
    if not(randomuclc_onlytext) or node.has_attribute(i,luatexbase.attributes.randuclcattr) then
      if math.random() < uclcratio then
        i.char = tex.uccode[i.char]
      else
        i.char = tex.lccode[i.char]
      end
    end
  end
  return head
end
```

10.14 randomchars

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

10.15 randomcolor and rainbowcolor

10.15.1 randomcolor – preliminaries

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

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

grey_lower = 0
grey_upper = 900

Rgb_lower = 1
rGb_lower = 1
```

```
rgB_lower = 1
Rgb_upper = 254
rGb_upper = 254
rgB_upper = 254
```

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

```
rainbow_step = 0.005
rainbow_Rgb = 1-rainbow_step -- we start in the red phase
rainbow_rGb = rainbow_step   -- values x must always be 0 < x < 1
rainbow_rgB = rainbow_step
rainind = 1          -- 1:red,2:yellow,3:green,4:blue,5:purple
```

This function produces the string needed for the pdf color stack. We need values 0]..[1 for the colors.

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

10.15.2 randomcolor – the function

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

Elsewise, all glyphs are taken.

```
randomcolor = function(head)
  for line in nodetraverseid(0,head) do
    for i in nodetraverseid(37,line.head) do
      if not(randomcolor_onlytext) or
         (node.has_attribute(i,luatexbase.attributes.randcolorattr))
      then
        color_push.data = randomcolorstring()  -- color or grey string
        line.head = nodeinsertbefore(line.head,i,nodecopy(color_push))
        nodeinsertafter(line.head,i,nodecopy(color_pop))
      end
    end
  end
  return head
end
```

10.16 randomerror

```
%
```

10.17 rickroll

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

10.18 substitutewords

This function is one of the rather usefull ones of this package. It replaces each occurance of one word by another word, which both are specified by the user. So nothing random or funny, but a real serious function! There are three levels for this function: At user-level, the user just specifies two strings that are passed to the function `addtosubstitutions`. This is needed as the # has a special meaning both in TEXs definitions and in Lua. In this second step, the list of substitutions is just extended, and the real work is done by the function `substiuteword` which is registered in the `process_input_buffer` callback. Once the substitution list is built, the rest is very simple: We just use gsub to substitute, do this for every item in the list, and that's it.

```
substitutewords_strings = {}

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

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

10.19 tabularasa

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

```
tabularasa_onlytext = false

tabularasa = function(head)
  local s = nodenew(nodeid"kern")
  for line in nodetraverseid(nodeid"hlist",head) do
    for n in nodetraverseid(nodeid"glyph",line.head) do
      if not(tabularasa_onlytext) or node.has_attribute(n,luatexbase.attributes.tabularasaattr) t
        s.kern = n.width
        nodeinsertafter(line.list,n,nodecopy(s))
        line.head = noderemove(line.list,n)
      end
    end
  end
  return head
end
```

10.20 uppercasecolor

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

```
uppercasecolor_onlytext = false

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

10.21 colorstretch

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

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

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

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

10.21.1 colorstretch – preliminaries

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

```
keeptext = true
colorexpansion = true

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


colorstretchnumbers = true
drawstretchthreshold = 0.1
drawexpansionthreshold = 0.9
```

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

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

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

    if colorexpansion then  -- if also the font expansion should be shown
      local g = line.head
        while not(g.id == 37) do
         g = g.next
        end
      exp_factor = g.width / f[g.char].width
      exp_color = colorstretch_coloroffset + (1-exp_factor)*10 .. " g"
      rule_bad.width = 0.5*line.width  -- we need two rules on each line!
    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 bett
    rule_bad.depth = tex.baselineskip.width*chickenize_rule_bad_depth

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

```

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

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

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

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

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

Now we are ready with the boxes and stuff and everything. However, a very useful information might be the amount of stretching, not encoded as color, but the real value. In concreto, I mean: narrow boxes get one color, loose boxes get another one, but only if the badness is above a certain amount. This information is printed into the right-hand margin. The threshold is user-adjustable.

```
    if colorstretchnumbers then
      j = 1
      glue_ratio_output = {}
      for s in string.utfvalues(math.abs(glue_ratio)) do -- using math.abs here gets us rid of the
        local char = unicode.utf8.char(s)
        glue_ratio_output[j] = nodenew(37,1)
        glue_ratio_output[j].font = font.current()
        glue_ratio_output[j].char = s
        j = j+1
      end
      if math.abs(glue_ratio) > drawstretchthreshold then
        if glue_ratio < 0 then color_push.data = "0.99 0 0 rg"
        else color_push.data = "0 0.99 0 rg" end
      else color_push.data = "0 0 0 rg"
      end

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

dubstepize

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

```

```

scorpionize

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

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

10.22 zebranize

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

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

10.22.1 zebranize – preliminaries

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

10.22.2 zebranize – the function

This code has to be revisited, it is ugly.

```
function zebranize(head)
  zebracolor = 1
  for line in nodetraverseid(nodeid"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(nodeid"glyph",line.head) do
      if n.next then else
        nodeinsertafter(line.head,n,nodecopy(color_pull))
      end
    end

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

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

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

And that's it! ☺

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

11 Drawing

A *very* first, experimental implementation of a drawing of a chicken. The parameters should be consistent, easy to change and that monster should look more like a cute chicken. However, it is chicken, it is Lua, so it belongs into this package. So far, all numbers and positions are hard coded, this will of course change!

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

function move (p)
  pdf_print(p[1],p[2],"m")
end

function line (p)
  pdf_print(p[1],p[2],"l")
end

function curve(p1,p2,p3)
  pdf_print(p1[1], p1[2],
            p2[1], p2[2],
            p3[1], p3[2], "c")
end

function close ()
  pdf_print("h")
end

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

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

```

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

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

function disturb_point(point)
  return {point[1] + math.random()*5 - 2.5,
          point[2] + math.random()*5 - 2.5}
end

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

  local right_end = disturb_point(right)

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

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

12 Known Bugs

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.

13 To Do's

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

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 differ between character and punctuation.

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

chickenmath chickenization of math mode

14 Literature

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

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

15 Thanks

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

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