The *cloze* package[*]

Josef Friedrich
josef@friedrich.rocks
github.com/Josef-Friedrich/cloze

v1.2 from 2016/06/23

[*]Many thanks to Robert-Michael Huber for his advice and to Paul Isambert for his article *"Three things you can do with LuaTEX that would be extremely painful otherwise"* in TUGboat, Volume 31 (2010), No. 3. This article helped a lot to write this package.

Contents

1 Introduction

cloze is a LaTeX package to generate cloze texts. It uses the capabilities of the modern TeX engine *LuaTeX*. Therefore, you must use LuaLaTeX to create documents containing gaps.

```
lualatex cloze-text.tex
```

The main feature of the package is that the formatting doesn't change when using the `hide` and `show` (→ 2.2.7) options.

> Lorem ipsum *dolor sit* amet, consectetur *adipisicing* elit, sed do eiusmod tempor incididunt ut labore et *dolore magna* aliqua. Ut enim ad minim veniam, quis nostrud *exercitation* ullamco laboris nisi ut *aliquip* ex ea commodo consequat.

The command `\clozeset{hide}` only shows gaps. When you put both texts on top of each other you will see that they perfectly match.

> Lorem ipsum __________ amet, consectetur ____________ elit, sed do eiusmod tempor incididunt ut labore et _______________ aliqua. Ut enim ad minim veniam, quis nostrud _____________ ullamco laboris nisi ut ________ ex ea commodo consequat.

2 Usage

There are three commands and one environment to generate cloze texts: `\cloze`, `\clozefix`, `\clozefil` and `clozepar`.

2.1 The commands and environments

2.1.1 `\cloze`

`\cloze` `\cloze[`⟨*options*⟩`]{`⟨*some text*⟩`}`: The command `\cloze` is similar to a command that offers the possibility to underline the texts. `\cloze` does not prevent line breaks. The width of a gap depends on the number of letters and the font used. The only option which affects the widths of a gap is the option `margin` (→ 2.2.9).

> Lorem ipsum *dolor* sit amet, *consectetur* adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore *magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi* ut aliquip ex ea commodo consequat.

It is possible to convert a complete paragraph into a 'gap'. But don't forget: There is a special environment for this: `clozepar` (→ 2.1.5).

> *Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.*

The command `\cloze` doesn't change the behavior of the hyphenation. Let's try some long German words:

es *Telekommunikationsüberwachung* geht *Unternehmenssteuerfortentwicklungsgesetz* *Abteilungsleiterin* *Oberkommisarin* auch *Fillialleiterin* kurz *Oberkommisarin* *Unternehmenssteuerfortentwicklungsgesetz* *Fillialleiterin* *Metzgermeisterin* in *Abteilungsleiterin* der *Oberkommisarin* *Hochleistungsflüssigkeitschromatographie* *Fillialleiterin* Kürze *Unternehmenssteuerfortentwicklungsgesetz* *Metzgermeisterin* liegt *Abteilungsleiterin* die *Metzgermeisterin* *Abteilungsleiterin* Würze *Oberkommisarin*

2.1.2 `\clozesetfont`

`\clozesetfont` The gap font can be changed by using the command `\clozesetfont`. `\clozesetfont` redefines the command `\clozefont` which contains the font definition. Thus, the command `\clozesetfont{\Large}` has the same effect as `\renewcommand{\clozefont}{\Large}`.

Excepteur sint occaecat cupidatat non proident.

Please do not put any color definitions in `\clozesetfont`, as it won't work. Use the option `textcolor` instead (→ 2.2.8).

`\clozesetfont{\ttfamily\normalsize}` changes the gap text for example into a normal sized typewriter font.

Excepteur `sint` occaecat `cupidatat` non proident.

2.1.3 `\clozefix`

`\clozefix` `\clozefix[`⟨*options*⟩`]{`⟨*some text*⟩`}`: The command `\clozefix` creates gaps with a fixed width. The clozes are default concering the width `2cm`.

Lorem ipsum dolor sit amet:
1. *consectetur*
2. *adipisicing*
3. *elit*

sed do eiusmod.

Gaps with a fixed width are much harder to solve.

Lorem ipsum dolor *sit* amet, *consectetur* adipisicing elit, sed do eiusmod tempor incididunt *ut* labore et dolore magna aliqua.

Using the option `align` you can make nice tabulars like this:

Composer	Life span
Joseph Haydn	*1723-1809*
Wolfgang Amadeus Mozart	*1756-1791*
Ludwig van Beethoven	*1770-1827*

2.1.4 `\clozefil`

`\clozefil` `\clozefil[`⟨*options*⟩`]{`⟨*some text*⟩`}`: The name of the command is inspired by `\hfil`, `\hfill`, and `\hfilll`. Only `\clozefil` fills out all available horizontal spaces with a line.

Lorem ipsum dolor sit amet, *consectetur adipisicing elit, sed do eiusmod.*
Ut enim *ad minim veniam* exercitation.

2.1.5 `clozepar`

`clozepar` `\begin{clozepar}[`⟨*options*⟩`]` ...*some text* ...`\end{clozepar}`: The environment `clozepar` transforms a complete paragraph into a cloze text. The options `align`, `margin` and `width` have no effect on this environment.

Lorem ipsum dolor sit amet, consectetur adipisicing elit ullamco laboris nisi.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum.
Excepteur sint occaecat cupidatat non proident.

2.1.6 `\clozeline`

`\clozeline` `\clozeline[`⟨*options*⟩`]`: To create a cloze line of a certain width, use the command `\clozeline`. The default width of the line is `2cm`. In combination with the other cloze commands you can create for example an irregular alignment of the cloze text.

```
Ut enim ad
\clozeline[width=1cm]\cloze{minim}\clozeline[width=3cm]
minim veniam
```

Ut enim ad *minim* minim veniam,

2.1.7 `\clozelinefil`

`\clozelinefil` `\clozelinefil[`⟨*options*⟩`]`: This command `\clozelinefil` fills the complete available horizontal space with a line. Moreover, `\clozelinefil` was used to create `\clozefil`.

Lorem

2.2 The options

2.2.1 Local and global options

The *cloze* package distinguishes between *local* and *global* options. Besides the possiblity to set *global* options in the `\usepackage[`⟨*global options*⟩`]{`⟨*cloze*⟩`}` declaration, the cloze package offers a special command to set *global* options: `\clozeset{`⟨*global options*⟩`}`

2.2.2 `\clozeset`

`\clozeset`

`\clozeset{`⟨*global options*⟩`}`: The command can set *global* options for each paragraph.

```
\clozeset{textcolor=red} Lorem \cloze{ipsum} dolor \par
\clozeset{textcolor=green} Lorem \cloze{ipsum} dolor
```

Lorem *ipsum* dolor
Lorem *ipsum* dolor

`\clozeset` does not change the options within a paragraph. As you can see in the example below the last `\clozeset` applies the color green for both gaps.

```
\clozeset{textcolor=red} Lorem \cloze{ipsum} dolor
\clozeset{textcolor=green} Lorem \cloze{ipsum} dolor
```

Lorem *ipsum* dolor Lorem *ipsum* dolor

2.2.3 `\clozereset`

`\clozereset`

`\clozereset`: The command resets all *global* options to the default values. It has no effect on the *local* options.

```
\clozeset{
  thickness=3mm,
  linecolor=yellow,
  textcolor=magenta,
  margin=-2pt
}
```

Very *silly* global *options*

```
\clozereset
```

Relax! We can reset *those* options.

2.2.4 `\clozeshow` and `\clozehide`

`\clozeshow`
`\clozehide`

`\clozeshow` and `\clozehide`: This commands are shortcuts for `\clozeset{`⟨*show*⟩`}` and `\clozeset{`⟨*hide*⟩`}`.

```
\clozehide
```

Lorem ____________ amet, consectetur _________ elit.

```
\clozeshow
```

Lorem *ipsum dolor sit* amet, consectetur *adipisicing* elit.

2.2.5 `align`

`[align=`⟨*left/center/right*⟩`]`: Only the macro `\clozefix` (→ 2.1.3) takes the option `align` into account. Possible values are `left`, `center` and `right`. This option only makes sense, if the width of the line is larger than the width of the text.

Lorem ipsum (`left`)
Lorem ipsum (`center`)
Lorem ipsum (`right`)

2.2.6 `distance`

`[distance=`⟨*dimen*⟩`]`: The option `distance` specifies the spacing between the baseline of the text and the gap line. The larger the dimension of the option `distance`, the more moves the line down. Negative values cause the line to appear above the baseline. The default value is `1.5pt`.

Lorem ipsum dolor sit amet. (`1.5pt`)
Lorem ipsum dolor sit amet. (`3pt`)
~~*Lorem ipsum dolor sit amet.*~~ (`-3pt`)

2.2.7 `hide` and `show`

`[hide]` and `[show]`: By default the cloze text is displayed. Use the option `hide` to remove the cloze text from the output. If you accidentally specify both options – `hide` and `show` – the last option "wins".

Lorem ipsum ____________, consectetur __________ elit. (`hide`)
Lorem ipsum *dolor sit amet*, consectetur *adipisicing* elit. (`show`)
Lorem ipsum ____________, consectetur __________ elit. (`show,hide`)
Lorem ipsum *dolor sit amet*, consectetur *adipisicing* elit. (`hide,show`)

2.2.8 `linecolor` and `textcolor`

`[linecolor=`⟨*color name*⟩`]` and `[textcolor=`⟨*color name*⟩`]`: Values for both color options are color names used by the xcolor package. To define your own color use the following command:

```
\definecolor{myclozecolor}{rgb}{0.1,0.4,0.6}
\cloze[textcolor=myclozecolor]{Lorem ipsum}
```

Lorem ipsum dolor sit amet, consectetur (`myclozecolor`)
Lorem ipsum dolor sit amet, consectetur (`red`)
Lorem ipsum dolor sit amet, consectetur (`green`)

You can use the same color names to colorize the cloze lines.

Lorem ipsum dolor sit amet, consectetur (`myclozecolor`)
Lorem ipsum dolor sit amet, consectetur (`red`)
Lorem ipsum dolor sit amet, consectetur (`green`)

2.2.9 `margin`

`[margin=`⟨*dimen*⟩`]`: The option `margin` indicates how far the line sticks up from the text. The option can be used with the commands `\cloze`, `\clozefix` and `\clozefil`. The default value of the option is `3pt`.

Lorem ipsum *dolor* sit amet. (`0pt`)
Lorem ipsum *dolor* sit amet. (`5mm`)
Lorem ipsum *dolor* sit amet. (`1cm`)
Lorem ipsum *dolor* sit amet. (`6em`)
Lorem ipsum*dolor*sit amet. (`-4pt`)

Is a punctation mark placed directly after a gap, then the line breaks after this punctation mark. Even the most large value of `margin` does not affect this behavior.

Lorem , *ipsum* . *dolor* ; *sit* : *amet* , *consectetur* .
adipisicing ; *elit* : *sed* , *do* . *eiusmod* ; *tempor* .

2.2.10 `thickness`

`[thickness=`⟨*dimen*⟩`]`: The option `thickness` indicates how thick the line is. The option `distance` (→ 2.2.6) is not affected by this option, because the bottom of the line moves down. The default value of this option is `0.4pt`.

Lorem *ipsum dolor sit* amet. (`0.01pt`)
Lorem *ipsum dolor sit* amet. (`1pt`)
Lorem *ipsum dolor sit* amet. (`2pt`)

2.2.11 `width`

`[width=`⟨*dimen*⟩`]`: The only command which can be changed by the option `width` is `\clozefix` (→ 2.1.3). The default value of the option is `2cm`.

Lorem *dolor* amet. (`3cm`)
Lorem *dolor* amet. (`5cm`)
Lorem *dolor* amet. (`7cm`)

2.3 Special application areas

2.3.1 The `tabbing` environment

```
\begin{tabbing}
col1 \hspace{1cm} \= col2 \hspace{1cm} \= col3 \hspace{1cm} \= col4 \\
\cloze{col1} \> \> \clozefix{col3} \\
\end{tabbing}
```

col1	col2	col3	col4
col1		*col3*	

2.3.2 The `picture` environment

```
\begin{picture}(320,100)
\put(80,25){\cloze{Lorem}}
\put(160,50){\clozefix{ipsum}}
\put(240,75){\clozefil{dolor}}
\end{picture}
```

dolor

ipsum

Lorem

2.3.3 The `tabular` environment

```
\begin{tabular}{l c}
\cloze{Lorem} & \cloze{ipsum} \\
\clozefix{amet} & \clozefix{consectetur} \\
\cloze{sed} & \cloze{do} \\
\end{tabular}
```

Lorem	*ipsum*
amet	*consectetur*
sed	*do*

3 Implementation

3.1 The file `cloze.sty`

This four packages are used to build *cloze*:

- fontspec is not necessarily required. When using LuaLaTeX it is good form to load it. Apart from this the package supplies helpful messages, when you compile a LuaLaTeX document with pdfLaTeX.
- luatexbase allows to register multiple Lua callbacks.
- kvoptions takes the handling of the options.
- xcolor is required to colorize the text and the line of a gap.

```
\RequirePackage{fontspec,luatexbase-mcb,kvoptions,xcolor}
```

Load the cloze lua module and put all return values in the variable `cloze`.

```
\directlua{
  cloze = require('cloze')
}
```

3.1.1 Internal macros

`\cloze@set@to@global` Set the Lua variable `registry.is_global` to `true`. All options are then stored in the variable `registry.global_options`.

```
\def\cloze@set@to@global{%
 \directlua{cloze.set_is_global(true)}%
}
```

`\cloze@set@to@local` First unset the variable `registry.local_options`. Now set the Lua variable `registry.is_global` to `false`. All options are then stored in the variable `registry.local_options`.

```
\def\cloze@set@to@local{%
  \directlua{
    cloze.unset_local_options()
    cloze.set_is_global(false)
  }%
}
```

`\cloze@set@option` `\cloze@set@option` is a wrapper for the Lua function `registry.set_option`. `\cloze@set@option[`⟨*key*⟩`]{`⟨*value*⟩`}` sets a key ⟨*key*⟩ to the value ⟨*value*⟩.

```
\def\cloze@set@option[#1]#2{%
  \directlua{cloze.set_option('#1', '#2')}%
}
```

`\cloze@color` Convert a color definiton name to a PDF colorstack string, for example convert the color name `blue` to the colorstack string `0 0 1 rg 0 0 1 RG`. The macro definition `\cloze@color{blue}` builds itself the macro `\color@blue`, which expands to the PDF colorstack string. The colorstack string is necessary to generate a PDF colorstack whatsit.

```
\def\cloze@color#1{\csname\string\color@#1\endcsname}
```

`\cloze@set@local@options` This macro is used in all cloze commands to handle the optional arguments. First it sets the option storage to local and then it commits the options to the package *kvoptions* via the macro `\kvsetkeys{CLZ}{}`.

```
\def\cloze@set@local@options#1{%
  \cloze@set@to@local%
  \kvsetkeys{CLZ}{#1}%
}
```

`\cloze@start@marker` At the begining `\cloze@start@marker` registers the required Lua callbacks. Then it inserts a whatsit marker which marks the begin of a gap.

```
\def\cloze@start@marker#1{%
  \strut\directlua{
    cloze.register('#1')
    cloze.marker('#1', 'start')
  }%
}
```

`\cloze@stop@marker` `\cloze@stop@marker` inserts a whatsit marker that marks the end of gap.

```
\def\cloze@stop@marker#1{%
  \strut\directlua{
    cloze.marker('#1', 'stop')
  }%
}
```

`\cloze@margin` `\cloze@margin` surrounds a text in a gap with two `kerns`.

```
\def\cloze@margin#1{%
  \directlua{cloze.margin()}%
  #1%
  \directlua{cloze.margin()}%
}
```

3.1.2 Options

cloze offers key-value pairs to use as options. For processing the key-value pairs we use the package kvoptions. To make all key-value pairs accessibly to Lua code, we use the declaration `\define@key{`⟨*CLZ*⟩`}{`⟨*option*⟩`}[`⟨*/*⟩`]{`⟨*...*⟩`}`. This declaration comes from the package keyval.

At start all values are declared as global options. At the Lua side all values are now stored in the `registry.global_options` table.

```
\cloze@set@to@global
```

We use the abbreviation `CLZ` for *cloze* as family name and prefix.

```
\SetupKeyvalOptions{
  family=CLZ,
  prefix=CLZ@
}
```

3.1.2.1 `align`

Please read the section (→ 2.2.5) how to use the option `align`. `align` affects only the command `\clozefix` (→ 2.1.3).

```
\DeclareStringOption{align}
\define@key{CLZ}{align}[]{\cloze@set@option[align]{#1}}
```

3.1.2.2 `distance`

Please read the section (→ 2.2.6) how to use the option `distance`.

```
\DeclareStringOption{distance}
\define@key{CLZ}{distance}[]{\cloze@set@option[distance]{#1}}
```

3.1.2.3 `hide`

If the option `hide` appears in the commands, `hide` will be set to *true* and `show` to *false* on the Lua side. Please read the section (→ 2.2.7) how to use the option `hide`.

```
\DeclareVoidOption{hide}{%
  \cloze@set@option[hide]{true}%
  \cloze@set@option[show]{false}%
}
```

3.1.2.4 `linecolor`

Please read the section (→ 2.2.8) how to use the option `linecolor`.

```
\DeclareStringOption{linecolor}
\define@key{CLZ}{linecolor}[]{%
  \cloze@set@option[linecolor]{\cloze@color{#1}}%
}
```

3.1.2.5 `margin`

Please read the section (→ 2.2.9) how to use the option `margin`.

```
\DeclareStringOption{margin}
\define@key{CLZ}{margin}[]{\cloze@set@option[margin]{#1}}
```

3.1.2.6 `show`

If the option `show` appears in the commands, `show` will be set to *true* and `true` to *false* on the Lua side. Please read the section (→ 2.2.7) how to use the option `show`.

```
\DeclareVoidOption{show}{%
  \cloze@set@option[show]{true}%
  \cloze@set@option[hide]{false}%
}
```

3.1.2.7 `textcolor`

Please read the section (→ 2.2.8) how to use the option `textcolor`.

```
\DeclareStringOption{textcolor}
\define@key{CLZ}{textcolor}[]{%
  \cloze@set@option[textcolor]{\cloze@color{#1}}%
}
```

3.1.2.8 `thickness`

Please read the section (→ 2.2.10) how to use the option `thickness`.

```
\DeclareStringOption{thickness}
\define@key{CLZ}{thickness}[]{\cloze@set@option[thickness]{#1}}
```

3.1.2.9 `width`

Please read the section (→ 2.2.11) how to use the option `width`. `width` affects only the command `\clozefix` (→ 2.1.3).

```
\DeclareStringOption{width}
\define@key{CLZ}{width}[]{\cloze@set@option[width]{#1}}
```

```
\ProcessKeyvalOptions{CLZ}
```

3.1.3 Public macros

All public macros are prefixed with `\cloze`.

`\clozeset` The usage of the command `\clozeset` is described in detail in section (→ 2.2.2).

```
\newcommand{\clozeset}[1]{%
  \cloze@set@to@global%
  \kvsetkeys{CLZ}{#1}%
}
```

`\clozereset` The usage of the command `\clozereset` is described in detail in section (→ 2.2.3).

```
\newcommand{\clozereset}{%
  \directlua{cloze.reset()}
}
```

`\clozeshow` The usage of the command `\clozeshow` is described in detail in section (→ 2.2.4).

```
\newcommand{\clozeshow}{%
  \clozeset{show}
}
```

`\clozehide` The usage of the command `\clozehide` is described in detail in section (→ 2.2.4).

```
\newcommand{\clozehide}{%
  \clozeset{hide}
}
```

`\clozefont` The usage of the command `\clozefont` is described in detail in section (→ 2.1.2).

```
\newcommand{\clozefont}{\itshape}
```

`\clozesetfont` The usage of the command `\clozesetfont` is described in detail in section (→ 2.1.2).

```
\newcommand{\clozesetfont}[1]{%
  \renewcommand{\clozefont}[1]{%
    #1%
  }%
}
```

`\cloze` The usage of the command `\cloze` is described in detail in section (→ 2.1.1).

```
\newcommand{\cloze}[2][]{%
  \cloze@set@local@options{#1}%
  \cloze@start@marker{basic}%
  {%
    \clozefont\relax%
    \cloze@margin{#2}%
  }%
  \cloze@stop@marker{basic}%
}
```

`\clozefix` The usage of the command `\clozefix` is described in detail in section (→ 2.1.3).

```
\newcommand{\clozefix}[2][]{%
  \cloze@set@local@options{#1}%
  \cloze@start@marker{fix}%
  {%
    \clozefont\relax%
    \cloze@margin{#2}%
  }%
  \cloze@stop@marker{fix}%
}
```

`clozepar` The usage of the environment `clozepar` is described in detail in section (→ 2.1.5).

```
\newenvironment{clozepar}[1][]%
{%
  \par%
  \cloze@set@local@options{#1}%
  \cloze@start@marker{par}%
  \clozefont\relax%
}%
{%
  \cloze@stop@marker{par}%
  \par%
  \directlua{cloze.unregister('par')}%
}
```

`\clozefil` The usage of the command `\clozefil` is described in detail in section (→ 2.1.4).

```
\newcommand{\clozefil}[2][]{%
  \cloze[#1]{#2}\clozelinefil[#1]%
}
```

`\clozeline` The usage of the command `\clozeline` is described in detail in section (→ 2.1.6).

```
\newcommand{\clozeline}[1][]{%
  \cloze@set@local@options{#1}%
  \directlua{cloze.line()}%
}
```

`\clozelinefil` The usage of the command `\clozelinefil` is described in detail in section (→ 2.1.7).

```
\newcommand{\clozelinefil}[1][]{%
  \cloze@set@local@options{#1}%
  \strut%
  \directlua{cloze.linefil()}%
  \strut%
}
```

3.2 The file `cloze.lua`

3.2.0.1 Initialisation of the function tables

It is good form to provide some background informations about this Lua module.

```
if not modules then modules = { } end modules ['cloze'] = {
  version   = '0.1',
  comment   = 'cloze',
  author    = 'Josef Friedrich, R.-M. Huber',
  copyright = 'Josef Friedrich, R.-M. Huber',
  license   = 'The LaTeX Project Public License Version 1.3c 2008-05-04'
}
```

`nodex` is a abbreviation for *node eXtended*.

```
local nodex = {}
```

All values and functions, which are related to the option management, are stored in a table called `registry`.

```
local registry = {}
```

I didn't know what value I should take as `user_id`. Therefore I took my birthday and transformed it to a large number.

```
registry.user_id = 3121978
registry.storage = {}
registry.defaults = {
  ['align'] = 'l',
  ['distance'] = '1.5pt',
  ['hide'] = false,
  ['linecolor'] = '0 0 0 rg 0 0 0 RG', -- black
  ['margin'] = '3pt',
  ['resetcolor'] = '0 0 0 rg 0 0 0 RG', -- black
  ['show_text'] = true,
  ['show'] = true,
  ['textcolor'] = '0 0 1 rg 0 0 1 RG', -- blue
  ['thickness'] = '0.4pt',
  ['width'] = '2cm',
}
registry.global_options = {}
registry.local_options = {}
```

All those functions are stored in the table `cloze` that are registered as callbacks to the pre and post linebreak filters.

```
local cloze = {}
```

In the status table are stored state information, which are necessary for the recursive cloze generation.

```
cloze.status = {}
```

The `base` table contains some basic functions. `base` is the only table of this Lua module that will be exported.

```
local base = {}
base.is_registered = {}
```

3.2.1 Node precessing (nodex)

All functions in this section are stored in a table called `nodex`. `nodex` is a abbreviation for *node eXtended*. The `nodex` table bundles all functions, which extend the built-in `node` library.

3.2.1.1 Color handling (color)

`create_colorstack`

Create a whatsit node of the subtype `pdf_colorstack`. `data` is a PDF colorstack string like `0 0 0 rg 0 0 0 RG`.

```
function nodex.create_colorstack(data)
  if not data then
    data = '0 0 0 rg 0 0 0 RG' -- black
  end
  local whatsit = node.new('whatsit', 'pdf_colorstack')
  whatsit.stack = 0
  whatsit.data = data
  return whatsit
end
```

`create_color`

`nodex.create_color()` is a wrapper for the function `nodex.create_colorstack()`. It queries the current values of the options `linecolor` and `textcolor`. The argument `option` accepts the strings `line`, `text` and `reset`.

```
function nodex.create_color(option)
  local data
  if option == 'line' then
    data = registry.get_value('linecolor')
  elseif option == 'text' then
    data = registry.get_value('textcolor')
  elseif option == 'reset' then
    data = nil
  else
    data = nil
  end
  return nodex.create_colorstack(data)
end
```

3.2.1.2 Line handling (line)

`create_line`

Create a rule node, which is used as a line for the cloze texts. The `depth` and the `height` of the rule are calculated form the options `thickness` and `distance`. The argument `width` must have the length unit *scaled points*.

```
function nodex.create_line(width)
  local rule = node.new(node.id('rule'))
  local thickness = tex.sp(registry.get_value('thickness'))
  local distance = tex.sp(registry.get_value('distance'))
  rule.depth = distance + thickness
  rule.height = - distance
  rule.width = width
  return rule
end
```

`insert_list`

Insert a `list` of nodes after or before the `current`. The `head` argument is optional. In some edge cases it is unfortately necessary. if `head` is omitted the `current` node is used. The argument `position` can take the values `'after'` or `'before'`.

```
function nodex.insert_list(position, current, list, head)
  if not head then
    head = current
  end
  for i, insert in ipairs(list) do
    if position == 'after' then
      head, current = node.insert_after(head, current, insert)
    elseif position == 'before' then
      head, current = node.insert_before(head, current, insert)
    end
  end
  return current
end
```

`insert_line`

Enclose a rule node (cloze line) with two PDF colorstack whatsits. The first colorstack node dyes the line, the seccond resets the color.

Node list:

Variable name	Node type	Node subtype	Parameter
`n.color_line`	`whatsit`	`pdf_colorstack`	Line color
`n.line`	`rule`		`width`
`n.color_reset`	`whatsit`	`pdf_colorstack`	Reset color

```
function nodex.insert_line(current, width)
```

```
  return nodex.insert_list(
    'after',
    current,
    {
      nodex.create_color('line'),
      nodex.create_line(width),
      nodex.create_color('reset')
    }
  )
end
```

write_line

This function enclozes a rule node with color nodes as it the function `nodex.insert_line` does. In contrast to `nodex.insert_line` the three nodes are appended to TeX's 'current list'. They are not inserted in a node list, which is accessed by a Lua callback.

Node list:

Variable name	Node type	Node subtype	Parameter
-	`whatsit`	`pdf_colorstack`	Line color
-	`rule`		`width`
-	`whatsit`	`pdf_colorstack`	Reset color

```
function nodex.write_line()
  node.write(nodex.create_color('line'))
  node.write(nodex.create_line(tex.sp(registry.get_value('width'))))
  node.write(nodex.create_color('reset'))
end
```

3.2.1.3 Handling of extendable lines (linefil)

create_linefil

This function creates a line which stretchs indefinitely in the horizontal direction.

```
function nodex.create_linefil()
  local glue = node.new('glue')
  glue.subtype = 100
  glue.stretch = 65536
  glue.stretch_order = 3
  local rule = nodex.create_line(0)
  rule.dir = 'TLT'
  glue.leader = rule
  return glue
end
```

write_linefil

The function `nodex.write_linefil` surrounds a indefinitely strechable line with color whatsits and puts it to TeX's 'current (node) list'.

```
function nodex.write_linefil()
  node.write(nodex.create_color('line'))
  node.write(nodex.create_linefil())
  node.write(nodex.create_color('reset'))
end
```

3.2.1.4 Kern handling (kern)

create_kern

This function creates a kern node with a given width. The argument `width` had to be specified in scaled points.

```
function nodex.create_kern(width)
  local kern = node.new(node.id('kern'))
  kern.kern = width
  return kern
end
```

strut_to_hlist

To make life easier: We add at the beginning of each hlist a strut. Now we can add line, color etc. nodes after the first node of a hlist not before - after is much more easier.

```
function nodex.strut_to_hlist(hlist)
  local n = {} -- node
  n.head = hlist.head
  n.kern = nodex.create_kern(0)
  n.strut = node.insert_before(n.head, n.head, n.kern)
  hlist.head = n.head.prev
  return hlist, n.strut, n.head
end
```

write_margin

Write kern nodes to the current node list. This kern nodes can be used to build a margin.

```
function nodex.write_margin()
  local kern = nodex.create_kern(tex.sp(registry.get_value('margin')))
  node.write(kern)
end
```

search_hlist

Search for a `hlist` (subtype `line`). Return false, if no `hlist` can be found.

```
function nodex.search_hlist(head)
  while head do
    if head.id == node.id('hlist') and head.subtype == 1 then
      return nodex.strut_to_hlist(head)
    end
    head = head.next
  end
  return false
end
```

3.2.2 Option handling (registry)

The table `registry` bundels functions that deal with option handling.

3.2.2.1 Marker processing (marker)

A marker is a whatsit node of the subtype `user_defined`. A marker has two purposes:

1. Mark the begin and the end of a gap.
2. Store a index number, that points to a Lua table, which holds some additional data like the local options.

`create_marker`

We create a user defined whatsit node that can store a number (type = 100). In order to distinguish this node from other user defined whatsit nodes we set the `user_id` to a large number. We call this whatsit node a marker. The argument `index` is a number, which is associated to values in the `registry.storage` table.

```
function registry.create_marker(index)
  local marker = node.new('whatsit','user_defined')
  marker.type = 100 -- number
  marker.user_id = registry.user_id
  marker.value = index
  return marker
end
```

`write_marker`

Write a marker node to TeX's current node list. The argument `mode` accepts the string values `basic`, `fix` and `par`. The argument `position`. The argument `position` is either set to `start` or to `stop`.

```
function registry.write_marker(mode, position)
  local index = registry.set_storage(mode, position)
  local marker = registry.create_marker(index)
  node.write(marker)
end
```

`is_marker`

This functions checks if the given node `item` is a marker.

```
function registry.is_marker(item)
  if item.id == node.id('whatsit')
    and item.subtype == node.subtype('user_defined')
    and item.user_id == registry.user_id then
    return true
  else
    return false
  end
end
```

`check_marker`

This functions tests, whether the given node `item` is a marker. The argument `item` is a node. The argument `mode` accepts the string values `basic`, `fix` and `par`. The argument `position` is either set to `start` or to `stop`.

```
function registry.check_marker(item, mode, position)
```

```
  local data = registry.get_marker_data(item)
  if data and data.mode == mode and data.position == position then
    return true
  else
    return false
  end
end
```

get_marker

registry.get_marker returns the given marker. The argument item is a node. The argument mode accepts the string values basic, fix and par. The argument position is either set to start or to stop.

```
function registry.get_marker(item, mode, position)
  local out
  if registry.check_marker(item, mode, position) then
    out = item
  else
    out = false
  end
  if out and position == 'start' then
    registry.get_marker_values(item)
  end
  return out
end
```

get_marker_data

registry.get_marker_data tests whether the node item is a marker. The argument item is a node of unspecified type.

```
function registry.get_marker_data(item)
  if item.id == node.id('whatsit')
    and item.subtype == node.subtype('user_defined')
    and item.user_id == registry.user_id then
    return registry.get_storage(item.value)
  else
    return false
  end
end
```

get_marker_values

First this function saves the associatied values of a marker to the local options table. Second it returns this values. The argument marker is a whatsit node.

```
function registry.get_marker_values(marker)
  local data = registry.get_marker_data(marker)
  registry.local_options = data.values
  return data.values
end
```

remove_marker

This function removes a given whatsit marker. It only deletes a node, if a marker is given.

```
function registry.remove_marker(marker)
  if registry.is_marker(marker) then node.remove(marker, marker) end
end
```

3.2.2.2 Storage functions (storage)

get_index

registry.index is a counter. The functions registry.get_index() increases the counter by one and then returns it.

```
function registry.get_index()
  if not registry.index then
    registry.index = 0
  end
  registry.index = registry.index + 1
  return registry.index
end
```

set_storage

registry.set_storage() stores the local options in the Lua table registry.storage. It returns a numeric index number. This index number is the key, where the local options in the Lua table are stored. The argument mode accepts the string values basic, fix and par.

```
function registry.set_storage(mode, position)
  local index = registry.get_index()
  local data = {
    ['mode'] = mode,
    ['position'] = position
  }
  data.values = registry.local_options
  registry.storage[index] = data
  return index
end
```

get_storage

The function registry.get_storage() retrieves values which belong to a whatsit marker. The argument index is a numeric value.

```
function registry.get_storage(index)
  return registry.storage[index]
end
```

3.2.2.3 Option processing (option)

set_option

This function stores a value value and his associated key key either to the global (registry.global_options) or to the local (registry.local_options) option table. The global boolean variable registry.local_options controls in which table the values are stored.

```
function registry.set_option(key, value)
  if value == '' or value == '\\color@ ' then
```

```
    return false
  end
  if registry.is_global == true then
    registry.global_options[key] = value
  else
    registry.local_options[key] = value
  end
end
```

set_is_global

registry.set_is_global() sets the variable registry.is_global to the value value. It is intended, that the variable takes boolean values.

```
function registry.set_is_global(value)
  registry.is_global = value
end
```

unset_local_options

This function unsets the local options.

```
function registry.unset_local_options()
  registry.local_options = {}
end
```

unset_global_options

registry.unset_global_options empties the global options storage.

```
function registry.unset_global_options()
  registry.global_options = {}
end
```

get_value

Retrieve a value from a given key. First search for the value in the local options, then in the global options. If both option storages are empty, the default value will be returned.

```
function registry.get_value(key)
  if registry.has_value(registry.local_options[key]) then
    return registry.local_options[key]
  end
  if registry.has_value(registry.global_options[key]) then
    return registry.global_options[key]
  end
  return registry.defaults[key]
end
```

get_value_show

The function registry.get_value_show() returns the boolean value true if the option show is true. In contrast to the function registry.get_value() it converts the string value ‘true’ to a boolean value.

```
function registry.get_value_show()
  if
    registry.get_value('show') == true
  or
```

```
    registry.get_value('show') == 'true'
  then
    return true
  else
    return false
  end
end
```

`has_value`

This function tests whether the value `value` is not empty and has a value.

```
function registry.has_value(value)
  if value == nil or value == '' or value == '\\color@ ' then
    return false
  else
    return true
  end
end
```

`get_defaults`

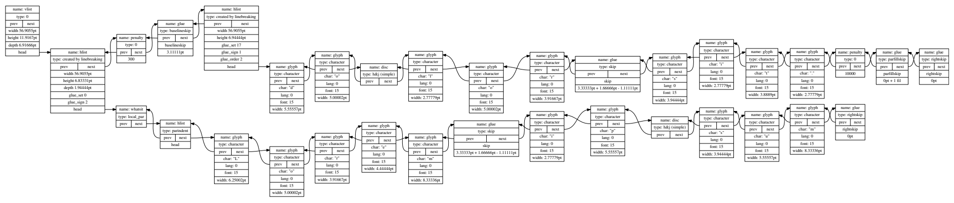
`registry.get_defaults(option)` returns a the default value of the given option.

```
function registry.get_defaults(option)
  return registry.defaults[option]
end
```

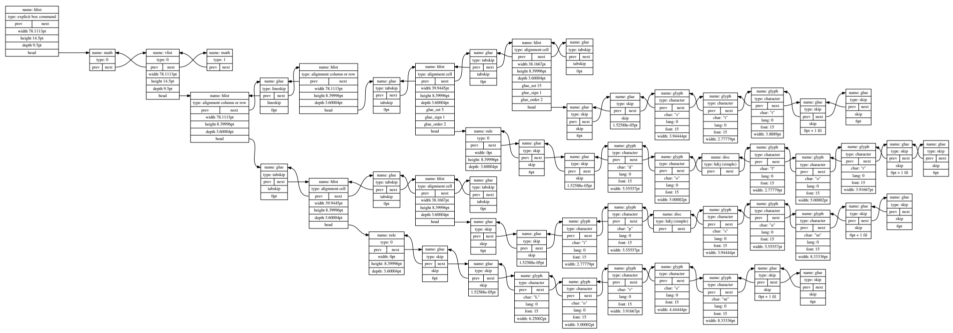
3.2.3 Assembly to cloze texts (cloze)

Some graphics for better understanding of the node tree:

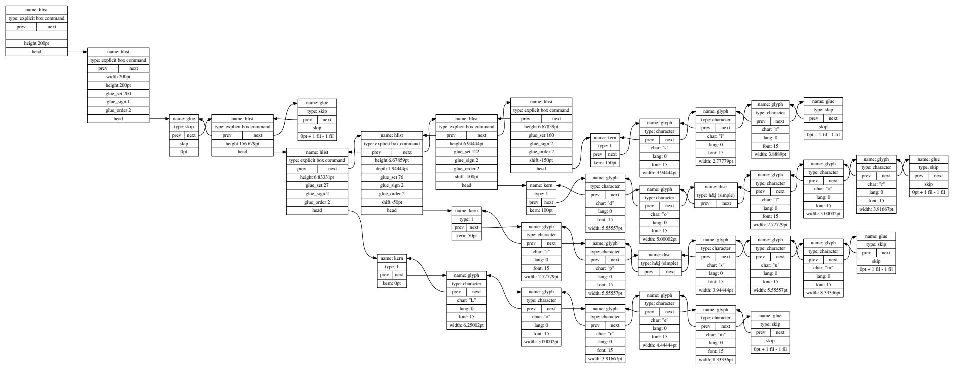
3.2.3.1 Paragraph



3.2.3.2 Tabular environment



3.2.3.3 Picture environment



`basic_make`

The function `cloze.basic_make()` makes one gap. The argument `start` is the node, where the gap begins. The argument `stop` is the node, where the gap ends.

```
function cloze.basic_make(start, stop)
  local n = {}
  local l = {}
  n.head = start
  if not start or not stop then
    return
  end
  n.start = start
  n.stop = stop
  l.width = node.dimensions(
    cloze.status.hlist.glue_set,
    cloze.status.hlist.glue_sign,
    cloze.status.hlist.glue_order,
    n.start,
    n.stop
  )
  n.line = nodex.insert_line(n.start, l.width)
  n.color_text = nodex.insert_list('after', n.line, {nodex.create_color('text')})
  if registry.get_value_show() then
    nodex.insert_list('after', n.color_text, {nodex.create_kern(-l.width)})
    nodex.insert_list('before', n.stop, {nodex.create_color('reset')}, n.head)
  else
    n.line.next = n.stop.next
    n.stop.prev = n.line.prev
  end
```

I some edge cases the lua callbacks get fired up twice. After the cloze has been created, the start and stop whatsit markers can be deleted.

```
  registry.remove_marker(n.start)
  registry.remove_marker(n.stop)
end
```

basic_search_stop

Search for a stop marker.

```
function cloze.basic_search_stop(head)
  local stop
  while head do
    cloze.status.continue = true
    stop = head
    if registry.check_marker(stop, 'basic', 'stop') then
      cloze.status.continue = false
      break
    end
    head = head.next
  end
  return stop
end
```

basic_search_start

Search for a start marker. Also begin a new cloze, if the boolean value `cloze.status.continue` is true. The knowledge of the last hlist node is a requirement to begin a cloze.

```
function cloze.basic_search_start(head)
  local start
  local stop
  local n = {}
  if cloze.status.continue then
    n.hlist = nodex.search_hlist(head)
    if n.hlist then
      cloze.status.hlist = n.hlist
      start = cloze.status.hlist.head
    end
  elseif registry.check_marker(head, 'basic', 'start') then
    start = head
  end
  if start then
```

```
    stop = cloze.basic_search_stop(start)
    cloze.basic_make(start, stop)
  end
end
```

`basic_recursion`

Parse recursivley the node tree. Start and stop markers can be nested deeply into the node tree.

```
function cloze.basic_recursion(head)
  while head do
    if head.head then
      cloze.status.hlist = head
      cloze.basic_recursion(head.head)
    else
      cloze.basic_search_start(head)
    end
      head = head.next
  end
end
```

`basic`

The corresponding LaTeX command to this lua function is `\cloze` (→ 2.1.1). The argument `head` is the head node of a node list.

```
function cloze.basic(head)
  cloze.status.continue = false
  cloze.basic_recursion(head)
  return head
end
```

`fix_length`

Calculate the length of the whitespace before (`l.kern_start`) and after (`l.kern_stopt`) the text.

```
function cloze.fix_length(start, stop)
  local l = {}
  l.width = tex.sp(registry.get_value('width'))
  l.text_width = node.dimensions(start, stop)
  l.align = registry.get_value('align')
  if l.align == 'right' then
    l.kern_start = - l.text_width
    l.kern_stop = 0
  elseif l.align == 'center' then
    l.half = (l.width - l.text_width) / 2
    l.kern_start = - l.half - l.text_width
    l.kern_stop = l.half
  else
    l.kern_start = - l.width
    l.kern_stop = l.width - l.text_width
  end
  return l.width, l.kern_start, l.kern_stop
end
```

fix_make

The function `cloze.fix_make` generates a gap of fixed width.

Node lists

Show text:

Variable name	Node type	Node subtype	Parameter
`n.start`	`whatsit`	`user_definded`	`index`
`n.line`	`rule`		`l.width`
`n.kern_start`	`kern`		Depends on `align`
`n.color_text`	`whatsit`	`pdf_colorstack`	Text color
	`glyphs`		Text to show
`n.color_reset`	`whatsit`	`pdf_colorstack`	Reset color
`n.kern_stop`	`kern`		Depends on `align`
`n.stop`	`whatsit`	`user_definded`	`index`

Hide text:

Variable name	Node type	Node subtype	Parameter
`n.start`	`whatsit`	`user_definded`	`index`
`n.line`	`rule`		`l.width`
`n.stop`	`whatsit`	`user_definded`	`index`

The argument `start` is the node, where the gap begins. The argument `stop` is the node, where the gap ends.

```
function cloze.fix_make(start, stop)
  local l = {} -- length
  local n = {} -- node
  l.width, l.kern_start, l.kern_stop = cloze.fix_length(start, stop)
  n.line = nodex.insert_line(start, l.width)
  if registry.get_value_show() then
    nodex.insert_list(
      'after',
      n.line,
      {
        nodex.create_kern(l.kern_start),
        nodex.create_color('text')
      }
    )
    nodex.insert_list(
      'before',
      stop,
      {
        nodex.create_color('reset'),
        nodex.create_kern(l.kern_stop)
      },
      start
    )
  else
    n.line.next = stop.next
  end
  registry.remove_marker(start)
  registry.remove_marker(stop)
end
```

`fix_recursion`

Function to recurse the node list and search after marker. `head` is the head node of a node list.

```
function cloze.fix_recursion(head)
  local n = {} -- node
  n.start, n.stop = false
  while head do
    if head.head then
      cloze.fix_recursion(head.head)
    else
      if not n.start then
        n.start = registry.get_marker(head, 'fix', 'start')
      end
      if not n.stop then
        n.stop = registry.get_marker(head, 'fix', 'stop')
      end
      if n.start and n.stop then
        cloze.fix_make(n.start, n.stop)
        n.start, n.stop = false
      end
    end
    head = head.next
  end
end
```

`fix`

The corresponding LaTeX command to this Lua function is `\clozefix` (→ 2.1.3). The argument `head` is the head node of a node list.

```
function cloze.fix(head)
  cloze.fix_recursion(head)
  return head
end
```

`par`

The corresponding LaTeX environment to this lua function is `clozepar` (→ 2.1.5).

Node lists

Show text:

Variable name	Node type	Node subtype	Parameter
`n.strut`	`kern`		width = 0
`n.line`	`rule`		`l.width` (Width from hlist)
`n.kern`	`kern`		`-l.width`
`n.color_text`	`whatsit`	`pdf_colorstack`	Text color
	`glyphs`		Text to show
`n.tail`	`glyph`		Last glyph in hlist
`n.color_reset`	`whatsit`	`pdf_colorstack`	Reset color

Hide text:

Variable name	Node type	Node subtype	Parameter
`n.strut`	`kern`		width = 0
`n.line`	`rule`		`l.width` (Width from hlist)

The argument head is the head node of a node list.

```
function cloze.par(head)
  local l = {} -- length
  local n = {} -- node
  for hlist in node.traverse_id(node.id('hlist'), head) do
    for whatsit in node.traverse_id(node.id('whatsit'), hlist.head) do
      registry.get_marker(whatsit, 'par', 'start')
    end
    l.width = hlist.width
    hlist, n.strut, n.head = nodex.strut_to_hlist(hlist)
    n.line = nodex.insert_line(n.strut, l.width)
    if registry.get_value_show() then
      nodex.insert_list(
        'after',
        n.line,
        {
          nodex.create_kern(-l.width),
          nodex.create_color('text')
        }
      )
      nodex.insert_list(
        'after',
        node.tail(head),
        {nodex.create_color('reset')}
      )
    else
      n.line.next = nil
    end
  end
  return head
end
```

3.2.4 Basic module functions (base)

register The base table contains functions which are published to the cloze.sty file. This function registers the functions cloze.par, cloze.basic and cloze.fix the Lua callbacks. cloze.par and cloze.basic are registered to the callback post_linebreak_filter and cloze.fix to the callback pre_linebreak_filter. The argument mode accepts the string values basic, fix and par.

```
function base.register(mode)
  local basic
  if mode == 'par' then
    luatexbase.add_to_callback(
      'post_linebreak_filter',
      cloze.par,
      mode
    )
    return true
  end
  if not base.is_registered[mode] then
    if mode == 'basic' then
      luatexbase.add_to_callback(
        'post_linebreak_filter',
        cloze.basic,
        mode
      )
    elseif mode == 'fix' then
      luatexbase.add_to_callback(
        'pre_linebreak_filter',
        cloze.fix,
        mode
      )
    else
      return false
    end
    base.is_registered[mode] = true
  end
end
```

unregister

`base.unregister(mode)` deletes the registered functions from the Lua callbacks. The argument `mode` accepts the string values `basic`, `fix` and `par`.

```
function base.unregister(mode)
  if mode == 'basic' then
    luatexbase.remove_from_callback('post_linebreak_filter', mode)
  elseif mode == 'fix' then
    luatexbase.remove_from_callback('pre_linebreak_filter', mode)
  else
    luatexbase.remove_from_callback('post_linebreak_filter', mode)
  end
end
```

Publish some functions to the `cloze.sty` file.

```
base.linefil = nodex.write_linefil
base.line = nodex.write_line
base.margin = nodex.write_margin
base.set_option = registry.set_option
base.set_is_global = registry.set_is_global
base.unset_local_options = registry.unset_local_options
base.reset = registry.unset_global_options
base.get_defaults = registry.get_defaults
base.marker = registry.write_marker
```

```
return base
```

Change History

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.