The luacode package

Manuel Pégourié-Gonnard <mpg@elzevir.fr>

v1.2a 2012/01/23

Abstract

Executing Lua code from within \TeX{} with \texttt{\directlua} can sometimes be tricky: there is no easy way to use the percent character, counting backslashes may be hard, and Lua comments don’t work the way you expect. This package provides the \texttt{\luadirect} command and the \texttt{\luacode(*)} environments to help with these problems, as well as helper macros and a debugging mode.

Contents

1 Documentation .......................... 1
  1.1 Lua code in \LaTeX{} .................. 1
  1.2 Helper macros .......................... 3
  1.3 Debugging .......................... 3

2 Implementation .................. 3
  2.1 Preliminaries .................. 4
  2.2 Internal code .................. 5
  2.3 Public macros and environments .................. 7

3 Test file .................. 8

1 Documentation

1.1 Lua code in \LaTeX{}

For an introduction to the most important gotchas of \texttt{\directlua}, see \texttt{lualatex-doc.pdf}. Before presenting the tools in this package, let me insist that the best way to manage a non-trivial piece of Lua code is probably to use an external file and source it from Lua, as explained in the cited document.

\texttt{\directlua} has changed along version of \LaTeX, so this package provides a \texttt{\luadirect} command which is an exact synonym of \texttt{\directlua} except that it doesn’t have the funny, changing parts of its syntax, and benefits from the debugging facilities described below (1.3).\footnote{And expands in two steps instead of one. If you don’t know what it means, then you hopefully don’t need to.}

The problems with \texttt{\directlua} (or \texttt{\luadirect}) are mainly with \LaTeX{} special characters. Actually, things are not that bad, since most special characters do work, namely: \_, \#, \$, \{, \}. Three are a bit tricky but they can be managed with \texttt{\string}: \, \# and ~. Only \% is really hard
to obtain. Also, \TeX\ macros are expanded, which is good since it allows to pass information
from \TeX\ to Lua, but you must be careful and use only purely expandable macros.

\section*{\texttt{\textbackslash luaexec}}

The \texttt{\textbackslash luaexec} command is similar to \texttt{\textbackslash luadirect} but with a few additional features:\footnote{And one major drawback: it is not purely expandable. See previous note.} \texttt{\textbackslash\textbackslash} gives a double backslash (see note below) \% a percent character, and \# just works. For single
backslashes, \texttt{\string} is still needed. Also, \TeX\ macros are expanded.

\section*{\texttt{\textbackslash luacode}}

The \texttt{\textbackslash luacode} environment is similar to \texttt{\textbackslash luaexec}, except that you can now use \% and \# directly
(but \% and \# also work) and the line breaks are respected, so that you can use line-wise Lua
comments in the normal way, without mistakenly commenting the rest of the chunk.

Only the backslashes and the braces keep their special meaning, so that macros still work as
usual, and you still need to use \texttt{\string} to get a single backslash.

\section*{\texttt{\textbackslash luacode*}}

The variant \texttt{\textbackslash luacode*} goes further and makes even backslash a normal character, so that you
don’t need any trick to obtain a single backslash. On the other end, macros don’t work any
more. So, the content of a \texttt{\textbackslash luacode*} is interpreted exactly as if it were in a normal Lua file,
directly fed to the Lua interpreter without any \TeX\ intervention.

The following table summarizes how to use special characters with the various commands
and environments.

<table>
<thead>
<tr>
<th></th>
<th>\texttt{\textbackslash luadirect}</th>
<th>\texttt{\textbackslash luaexec}</th>
<th>\texttt{\textbackslash luacode}</th>
<th>\texttt{\textbackslash luacode*}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Single backslash</td>
<td>\texttt{\string\textbackslash}</td>
<td>\texttt{\string\textbackslash}</td>
<td>\texttt{\string\textbackslash}</td>
<td>Just \texttt{\textbackslash}</td>
</tr>
<tr>
<td>Double backslash</td>
<td>\texttt{\string\textbackslash\textbackslash}</td>
<td>\texttt{\string\textbackslash\textbackslash}</td>
<td>\texttt{\string\textbackslash\textbackslash}</td>
<td>\texttt{\string\textbackslash\textbackslash}</td>
</tr>
<tr>
<td>Tilde</td>
<td>\texttt{\string~}</td>
<td>\texttt{\string~}</td>
<td>\texttt{\string~}</td>
<td>\texttt{\string~}</td>
</tr>
<tr>
<td>Sharp</td>
<td>\texttt{\string#}</td>
<td>\texttt{#}</td>
<td>\texttt{#} (or \texttt{#})</td>
<td>\texttt{#}</td>
</tr>
<tr>
<td>Percent</td>
<td>No easy way</td>
<td>\texttt{%}</td>
<td>\texttt{%} (or \texttt{%})</td>
<td>\texttt{%}</td>
</tr>
<tr>
<td>\TeX\ comments</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Lua line comments</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\section*{Backslashes and Lua strings}

In the table and descriptions above, “double backslash” means that the Lua interpreter will see a double backslash. It may then turn it into a single backslash in the context of a Lua string delimited by single or double quotes as opposed to a Lua string delimited by brackets, see \textit{Programming in Lua} section 2.4. Similarly, a single backslash may or
may not be interpreted as starting an escape sequence. For example:

\begin{verbatim}
\begin{luacode}
a = "\" -- a contains a single backslash
b = [[\\]] -- b contains two backslashes
c = "\\\" -- c contains two backslashes too
d = "line one\newline two" -- d contains a newline character
e = [[single\newline]] -- e contains no newline character
\end{luacode}
\end{verbatim}

The alert reader may notice that in the case of \texttt{\textbackslash luadirect} and \texttt{\textbackslash luaexec}, single backslashes are a bit weird. For example with

\texttt{\textbackslash luaexec{\texio.write_nl("line one\string\newline two")}}
\TeX{} will see \texttt{\textbackslash \textbackslash \text{line}} as a control sequence which is the “argument” of \texttt{\textbackslash \text{string}} and the Lua interpreter will consider only \textbackslash \text{\textbackslash} as an escape sequence, and \texttt{\textbackslash \text{line}} as independent characters. In practice, this should not have any unwanted consequences (except perhaps on the sanity of the reader).

\textbf{luacodestar} Technical notes on environments. The environments will not work inside the argument of a command (just as with verbatim commands and environments). Also, you are supposed to leave a space (or end-of-line) after the \texttt{\textbackslash \text{begin}}\texttt{\textbackslash \text{luacode}} or \texttt{\textbackslash \text{begin}}\texttt{\textbackslash \text{luacode}}*, which is probably a natural thing to do anyway. Finally, if you wish to define derived environments, you’ll need to use \texttt{\textbackslash \text{luacode}} \ldots \texttt{\textbackslash \text{end}}\texttt{\textbackslash \text{luacode}} instead of the usual \texttt{\textbackslash \text{begin}} \texttt{\textbackslash \text{end}} pair in your environment’s definition. For the stared variant, use \texttt{\textbackslash \text{luacodestar}} and \texttt{\textbackslash \text{endluacodestar}}.

The test file (section 3, or \texttt{test-luacode.tex} in the same directory as this document) provides stupid but complete examples.

1.2 Helper macros

As mentioned in the previous section, except for trivial pieces of codes (or examples) it is good practice to keep all your Lua code in separate .\texttt{\text{lua}} files and then use \texttt{\textbackslash \text{luadirect}} only to \texttt{\text{require}()} or \texttt{\text{dofile}()} it and define \LaTeX{} wrappers for some functions, eg:

\begin{verbatim}
\newcommand*{\foo}[2]{\texttt{\textbackslash \text{luadirect}}\{\texttt{\textbackslash \text{foo}}("1", \ #2)\}}
\end{verbatim}

This way, problems with \TeX{} special characters are avoided, since most of the Lua is never seen by \TeX{}. Unfortunately, there is still potential for problems. For example \texttt{\textbackslash \text{foo}{\"a\"b}{2}} will cause the Lua interpreter to complain since the ” in \texttt{\textbackslash \text{#1}} will end the string; we want the Lua interpreter to see ”a\"b” as the first argument.

\textbf{\texttt{\textbackslash \text{luastring}}} Fortunately, \texttt{\textbackslash \text{\textbackslash \text{luastring}}} offers a primitive that does exactly what we need: escape characters that need to be escaped in a Lua string. Unfortunately, it has a very long name (especially in the prefixed form available in \TeX{}): \texttt{\textbackslash \text{luatexluaescapestring}}. Also, you need to think to use quotes in addition to this primitive. So this package provides a shorter version: \texttt{\textbackslash \text{luastring}} that also include the quotes, so a safer version of \texttt{\textbackslash \text{foo}} might be defined as

\begin{verbatim}
\newcommand*{\foo}[2]{\texttt{\textbackslash \text{luadirect}}\{\texttt{\textbackslash \text{luastring}}\{\texttt{\textbackslash \text{#1}}\}, \ #2\}}
\end{verbatim}

It should be noted that the argument of \texttt{\textbackslash \text{luastring}} is fully expanded\footnote{If you don’t know what this means, just skip this paragraph.} before being turned into a Lua string. In case where such an expansion is unwanted, two variants are provided: \texttt{\textbackslash \text{luastring}} for no expansion, and \texttt{\textbackslash \text{luastring}} for one-level expansion (of the first token) only.

1.3 Debugging

The commands \texttt{\textbackslash \text{luadirect}} and \texttt{\textbackslash \text{luaexec}} as well as the environments \texttt{\textbackslash \text{luacode}} and \texttt{\textbackslash \text{luacode}}* can optionally print the Lua code as it will be seen by the Lua interpreter in the log file before executing it. The feature is disabled by default and can be turned on and off using \texttt{\textbackslash \text{LuaCodeDebugOn}} and \texttt{\textbackslash \text{LuaCodeDebugOff}} (which obey the usual \TeX{} scoping rules).

2 Implementation
2.1 Preliminaries

Catcode defenses.

\begingroup\catcode61\catcode48\catcode32=10\relax% = and space
\catcode123 1 % {
\catcode125 2 % }
\catcode 35 6 % #
\toks0{\endlinechar\the\endlinechar}%
\edef\x{\endlinechar13}%
\def\y#1 #2 {%
\toks0\expandafter{\the\toks0 \catcode#1 \the\catcode#1}%
\edef\x{\x \catcode#1 #2}}%
\edef\x{\x #1 \catcode#1 #2}%
\y 13 5 % carriage return
\y 61 12 % =
\y 32 10 % space
\y 123 1 % {
\y 125 2 % }
\y 35 6 % #
\y 64 11 % @ (letter)
\y 39 12 % ‘
\y 42 12 % *
\y 45 12 % −
\y 46 12 % .
\y 47 12 % /
\y 91 12 % [
\y 93 12 % ]
\y 94 7 % ^
\y 96 12 % ‘
\y 126 13 % −
\toks0\expandafter{\the\toks0 \relax\noexpand\endinput}%
\edef\y#1{\noexpand\expandafter\endgroup%
\noexpand\ifx#1\relax \edef#1{\the\toks0}\x\relax%
\noexpand\else \noexpand\expandafter\noexpand\endinput%
\noexpand\fi}%
\expandafter\y\csname luacode@sty@endinput\endcsname%

Package declaration.
\ProvidesPackage{luacode}[2012/01/23 v1.2a lua-in-tex helpers (mpg)]

Make sure LuaTEX is used.
\RequirePackage{ifluatex}
\ifluatex\else
\PackageError{luacode}{LuaTeX is required for this package. Aborting.}{%This package can only be used with the LuaTeX engine, Package loading has been stopped to prevent additional errors.}
\expandafter\luacode@sty@endinput
\fi

Use luatexbase for catcode tables.
\RequirePackage{luatexbase}


2.2 Internal code

Produce Lua code printing debug info for the given argument.

```latex
\newcommand \luacode@printdbg [1] {%
  \texio.write_nl('log',
  '-- BEGIN lua code debug (on input line \the\inputlineno)')
  \texio.write_nl('log', "\luatexluaescapestring{#1}")
  \texio.write_nl('log',
  '-- END lua code debug (on input line \the\inputlineno)')
}%
```

Execute a piece of Lua code, possibly printing debug info. `maybe@printdbg` will be either `printdbg` or `gobble`, see user macros.

```latex
\newcommand \luacode@dbg@exec [1] {%
  \directlua {\luacode@maybe@printdbg{#1}
  #1}%\}
}%
```

Execute a piece of code, with shortcuts for double-backslash, percent and tilde, and trying to preserve newlines. This internal macro is long so that we can use in the environment, while the corresponding user command will be short. Make sure ~ is active.

```latex
\begingroup \catcode'~\active \expandafter\endgroup\@firstofone{%
\newcommand \luacode@execute [1] {%
\begingroup
\escapechar92
\newlinechar10
\edef\{\{\string\}\}\
\edef\{-\{\string-\}\
\let\%=\luacode@percentchar
\let\#=\luacode@sharpchar
\expandafter\expandafter\expandafter\endgroup
\luacode@dbg@exec{#1}}%
}%
```

Catcode 12 percent and sharp characters for use in the previous command.

```latex
\begingroup \catcode\m@ne \edef\aux{\endgroup\unexpanded{\newcommand\luacode@percentchar}{\string\%}\
\unexpanded{\newcommand\luacode@sharpchar }{\string\#}\
\aux}
```

Generic code for environments; the argument is the name of a catcode table. We’re normally inside a group, but let’s open a new one in case we’re called directly rather that using `\begin`. Define the end marker to be `\end{<envname>}` with current catcodes.

```latex
\newcommand*\luacode@begin [1] {%
\begingroup
\escapechar92
\luatexcatcodetable#1\relax
\edef\luacode@endmark{\string\end{\@currenvir}}%
\expandafter\def \expandafter\luacode@endmark \expandafter{\luatexscantextokens \expandafter{\luacode@endmark}}%
\luacode@grab@body}
```
We'll define the body grabber in a moment, but let's see how the environment ends now.

\newcommand\luacode@end{%
  \edef\luacode@next{%
    \noexpand\luacode@execute{\the\luacode@lines}%
  }
  \expandafter\endgroup
}

It is not possible to grab the body using a macro with delimited argument, since the end marker may contains open-group characters, depending on the current catcode regime. So we collect it linewise and check each line against the end marker.

Storage for lines.

\newtoks\luacode@lines
\newcommand*\luacode@addline [1] {\luacode@lines{\the\luacode@lines#1^^J}}

Loop initialisation. Set endlinechar explicitely so that we can use it as a delimiter (and later when writing the code to Lua). Eat up the first token which is supposed to be a (catcode 12) \endlinechar character token.

\newcommand \luacode@grab@body [1] {\luacode@lines{}\endlinechar10\luacode@grab@lines}

The actual line-grabbing loop.

\long\def\luacode@grab@lines#1^^J{\def\luacode@curr{#1}\luacode@strip@spaces
  \ifx\luacode@curr\luacode@endmark
    \expandafter\luacode@end
  \else
    \expandafter\luacode@addline\expandafter{\luacode@curr} \expandafter\luacode@grab@lines
  \fi}

Strip catcode 12 spaces from the beginning of the token list inside \luacode@curr. First we need catcode 12 space, then we proceed in the usual way.
Finally, we need a custom catcode table for the default environment: everything other, except backslash, braces and letters which retain their natural catcodes. Be careful about the name of the macro for setting catcode ranges which is currently changing in luatexbase. The group here doesn’t matter since catcode table settings are always global.

\newluatexcatcodetable \luacode@table@soft
\begingroup
\ifdefined\SetCatcodeRange \else
   \let\SetCatcodeRange\setcatcoderange
\fi
\setluatexcatcodetable \luacode@table@soft {%
   \luatexcatcodetable\CatcodeTableOther
   \catcode 92 0
   \catcode 123 1
   \catcode 125 2
   \SetCatcodeRange {65}{90} {11}
   \SetCatcodeRange {97}{122}{11}
}
\endgroup

2.3 Public macros and environments

Debugging.
\newcommand \LuaCodeDebugOn {\let \luacode@maybe@printdbg \luacode@printdbg}
\newcommand \LuaCodeDebugOff {\let \luacode@maybe@printdbg \@gobble}
\LuaCodeDebugOff

The \luadirect and \luaexec macros.
\ifdefinable\luadirect{\let\luadirect\luaexec[\luacode@exec]}
\newcommand*\luaexec [1] {\luacode@execute{#1}}

Environments using different catcode tables.
\newenvironment {luacode} {\luacode@begin\luacode@table@soft} {}
\newenvironment {luacode*} {\luacode@begin\CatcodeTableOther} {}
\newcommand \luacodestar {\@nameuse{luacode*}}
\def \endluacodestar {\@nameuse{endluacode*}}

Helper macros
\newcommand \luastring [1]{\"\luatexluaescapestring{#1}\"}
\newcommand \luastringO [1]{\luastring{\unexpanded{\expandaftersavequote{#1}}}}
\newcommand \luastringN [1]{\luastring{\unexpanded{#1}}}

We’re already done!
\luacode@sty@endinput
⟨/texpackage⟩
3 Test file

TODO: this test files requires manual checking that the output (pdf and log file) is correct; this should be fixed.

154 ⟨testlatex⟩
155 \documentclass{minimal}
156 \usepackage{luacode}
157 \begin{document}
158
159 \newcommand\foo{3}
160
161 ⟨
162 \textdirect{
163 \texio.write_nl("Special chars: _ ^ & $ { } working.\string
164 .. "Backslashes need a bit of care.\string
165 .. "Sharps and tildes too: # doubled, but \string# and \string-
166 % a tex comment: no easy way to get a %
167 \tex.sprint("\string\pi \string\neq", tostring(math.pi))
168 % we can use TeX macros
169 \tex.sprint("\string\pi \string\neq", math.sqrt(\foo))
170 }
171 ⟩
172
173 ⟨
174 \texexec{
175 \texio.write_nl("Special chars: _ ^ & $ { } working.\string
176 .. "Backslashes still need a bit of care.\string
177 .. "Single sharps are easier now: \#"
178 % a tex comment: we also get a % below
179 \tex.sprint("\pi \neq", tostring(math.pi):gsub(’%.’, ’’))
180 % we can use TeX macros
181 \tex.sprint("\pi \neq", math.sqrt(\foo))
182 }
183 ⟩
184
185 ⟨
186 \begin{luacode}
187 \texio.write_nl("Special chars: _ ^ & $ { } working.\string
188 .. "Only backslashes still need a bit of care.\string
189 -- a lua comment: we could use \% below, too
190 \tex.sprint("\pi \neq", tostring(math.pi):gsub(’%.’, ’’))
191 -- we can use TeX macros
192 \tex.sprint("\pi \neq", math.sqrt(\foo))
193 \end{luacode}
194 ⟩
195
196 ⟨
197 \begin{luacode*}
198 \texio.write_nl("Special chars: _ ^ & $ { } - # \ working.\n"
199 .. "Only backslashes still need a bit of care.\string
200 -- a lua comment: the backlash is doubled as in normal Lua code
201 \tex.sprint("\pi \neq", tostring(math.pi):gsub(’%.’, ’’))
202 -- no way to use a TeX variable here
203 ⟩
Now track spurious spaces. This is the only part that is automatically checked, using grep in the Makefile.
\begin{luacode}
local foo
\end{luacode}
\begin{luacode*}
local foo
\end{luacode*}