The luacode package

Manuel Pégourié-Gonnard <mpg@elzevir.fr>

v1.2a 2012/01/23

Abstract

Executing Lua code from within TEX with `\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 `\luaexec` command and the `luacode(*)` environments to help with these problems, as well as helper macros and a debugging mode.

Contents

1 Documentation

1.1 Lua code in LATEX

For an introduction to the most important gotchas of `\directlua`, see `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.

`\luadirect` First, the exact syntax of `\directlua` has changed along version of LuaTEX, so this package provides a `\luadirect` command which is an exact synonym of `\directlua` except that it doesn't have the funny, changing parts of its syntax, and benefits from the debugging facilities described below (1.3).[1]

The problems with `\directlua` (or `\luadirect`) are mainly with TEX 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 `\string`: `\`, `#` and `~`. Only `%` is really hard

---

[1] And expands in two steps instead of one. If you don't know what it means, then you hopefully don't need to.

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.

`\luaexec` The `\luaexec` command is similar to `\luadirect` but with a few additional features:[2] `\\` gives a double backslash (see note below) `\%` a percent character, and `~` just works. For single backslashes, `\string` is still needed. Also, TEX macros are expanded.

`luacode` The `luacode` environment is similar to `\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 backslash and the braces keep their special meaning, so that macros still work as usual, and you still need to use `\string` to get a single backslash.

`luacode*` The variant `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 `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.

	`\luadirect`	`\luaexec`	`luacode`	`luacode*`
Macros	Yes	Yes	Yes	No
Single backslash	`\string\`	`\string\`	`\string\`	Just `\`
Double backslash	`\string\\`	`\\`	`\\`	`\\`
Tilde	`\string~`	`~`	`~`	`~`
Sharp	`\string#`	`\#`	`#` (or `\#`)	`#`
Percent	No easy way	`\%`	`%` (or `\%`)	`%`
TEX comments	Yes	Yes	No	No
Lua line comments	No	No	Yes	Yes

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 *Programming in Lua* section 2.4. Similarly, a single backslash may or may not be interpreted as starting an escape sequence. For example:

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

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

```
\luaexec{texio.write_nl("line one\string\nline two")}
```

---

[2] And one major drawback: it is not purely expandable. See previous note.

TeX will see `\nline` as a control sequence which is the “argument” of `\string` and the Lua interpreter will consider only `\n` as an escape sequence, and `line` as independent characters. In practice, this should not have any unwanted consequences (except perhaps on the sanity of the reader).

`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 `\begin{luacode}` or `\begin{luacode*}`, which is probably a natural thing to do anyway. Finally, if you wish to define derived environments, you’ll need to use `\luacode` ... `\endluacode` instead of the usual `\begin` `\end` pair in your environment’s definition. For the stared variant, use `\luacodestar` and `\endluacodestar`.

The test file (section 3, or `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 `.lua` files and then use `\luadirect` only to `require()` or `dofile()` it and define LaTeX wrappers for some functions, eg:

```
\newcommand*\foo[2]{\luadirect{foo("#1", #2)}}
```

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 `\foo{a"b}{2}` will cause the Lua interpreter to complain since the `"` in `#1` will end the string; we want the Lua interpreter to see `"a\"b"` as the first argument.

`\luastring` Fortunately, LuaTeX 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 LaTeX): `\luatexluaescapestring`. Also, you need to think to use quotes in addition to this primitive. So this package provides a shorter version: `\luastring` that also include the quotes, so a safer version of `\foo` might be defined as

```
\newcommand*\foo[2]{\luadirect{foo(\luastring{#1}, #2)}}
```

`\luastringN` `\luastringO` It should be noted that the argument of `\luastring` is fully expanded[3] before being turned into a Lua string. In case where such an expansion is unwanted, two variants are provided: `\luastringN` for no expansion, and `\luastringO` for one-level expansion (of the first token) only.

1.3 Debugging

`\LuaCodeDebugOn` `\LuaCodeDebugOff` The commands `\luadirect` and `\luaexec` as well as the environments `luacode` and `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 `\LuaCodeDebugOn` and `\LuaCodeDebugOff` (which obey the usual TeX scoping rules).

2 Implementation

```
⟨*texpackage⟩
```

---

[3] If you don’t know what this means, just skip this paragraph.

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}}%
  \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  40 12 % (
  \y  41 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\MessageBreak
    (command `lualatex'). Package loading has been stopped\MessageBreak
    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.

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

```
\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.

```
\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.

```
\begingroup \escapechar\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.

```
\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}%
    \noexpand\end{\@currenvir}}%
  \expandafter\endgroup
  \luacode@next}
```

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\expandafter{\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 procede in the usual way.

```
\begingroup\catcode32 12 \expandafter\endgroup
\@firstofone{\newcommand\luacode@spaceother{ }}
\newcommand \luacode@strip@spaces {%
  \expandafter\luacode@strip@sp@peek\luacode@curr\@nil}
\newcommand \luacode@strip@sp@peek {%
  \futurelet\@let@token\luacode@strip@sp@look}
\newcommand \luacode@strip@sp@look {%
  \expandafter\ifx\luacode@spaceother\@let@token
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi{%
    \afterassignment\luacode@strip@sp@peek
    \let\@let@token=
  }{%
    \luacode@strip@sp@def
  }}
\@ifdefinable \luacode@strip@sp@def \relax
\def \luacode@strip@sp@def #1\@nil{%
  \def\luacode@curr{#1}}
```

Finally, we need a custom catcode table for the default environment: everything other, except backslash, braces and letters which retain their natural catcodes.

Be carefull 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\luacode@dbg@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\expandafter{#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.

```
⟨*testlatex⟩
\documentclass{minimal}
\usepackage{luacode}
\begin{document}

\newcommand\foo{3}

\(
  \luadirect{
    texio.write_nl("Special chars: _ ^ & $ { } working.\string\n"
    .. "Backslashes need a bit of care.\string\n"
    .. "Sharps and tildes too: # doubled, but \string# and \string~")
    % a tex comment: no easy way to get a %
    tex.sprint("\string\\pi \string\\neq", tostring(math.pi))
    % we can use TeX macros
    tex.sprint("-", math.sqrt(\foo))
  }
\)


\(
  \luaexec{
    texio.write_nl("Special chars: _ ^ & $ { } ~ working.\string\n"
    .. "Backslashes still need a bit of care.\string\n"
    .. "Single sharps are easier now: \#")
    % a tex comment: we also get a % below
    tex.sprint("\\pi \\neq ", tostring(math.pi):gsub('\%.', '+'))
    % we can use TeX macros
    tex.sprint("-", math.sqrt(\foo))
  }
\)

\[
  \begin{luacode}
    texio.write_nl("Special chars: _ ^ & $ { } ~ # % working.\string\n"
    .. "Only backslashes still need a bit of care.\string\n")
    -- a lua comment: we could use \% below, too
    tex.sprint("\\pi \\neq ", tostring(math.pi):gsub('%.', '+'))
    -- we can use TeX macros
    tex.sprint("-", math.sqrt(\foo))
  \end{luacode}
\]

\[
  \begin{luacode*}
    texio.write_nl("Special chars: _ ^ & $ { } ~ # % \\ working.\n")
    -- a lua comment: the backlash is doubled as in normal Lua code
    tex.sprint("\\pi \\neq ", tostring(math.pi):gsub('%.', '+'))
    -- no way to use a TeX variable here
  \end{luacode*}
\]

\newenvironment{mathluacode} { \[ \luacode     }{ \endluacode     \] }
\newenvironment{mathluacode*}{ \[ \luacodestar }{ \endluacodestar \] }

\begin{mathluacode}
  local foo = "A full line.\string\n"
  tex.sprint("\\pi \\neq ", tostring(math.pi):gsub('%.', '+'))
  -- a lua comment: we could have used \% above, too
  tex.sprint("-", math.sqrt(\foo))
\end{mathluacode}

\begin{mathluacode*}
  local foo_bar = "A full line.\n"
  tex.sprint("\\pi \\neq ", tostring(math.pi):gsub('%.', '+'))
  -- a lua comment: no way to use a TeX variable here
\end{mathluacode*}

\begin{luacode*}
  function myfunc(str)
    assert(type(str) == 'string')
    tex.sprint(-2, str)
  end
\end{luacode*}

\newcommand*\mymac [1] {\texttt{\luadirect{myfunc(\luastring{#1})}}\par}
\mymac{abc}
\mymac{123}
\mymac{a"b\string\nc'd}

\def\mac{\onelevel}
\def\onelevel{fully expanded}
string : \texttt{\luadirect{myfunc(\luastring \mac)}}\par
stringO: \texttt{\luadirect{myfunc(\luastringO\mac)}}\par
stringN: \texttt{\luadirect{myfunc(\luastringN\mac)}}\par

\LuaCodeDebugOn
\luadirect  {local foo = 'bar'      .. \luastring{a"b'c}}
\luaexec    {local foo = 'bar\%\#'  .. \luastring{a"b'c}}
\begin{luacode}
  local foo = 'bar'
  local baz = 12 % 2
  assert(\luastring\mac  == 'fully expanded')
  -- assert(\luastringO\mac == '\\onelevel')
  -- assert(\luastringN\mac == '\\mac')
\end{luacode}

\LuaCodeDebugOff
\luadirect{local rem = 'dbg should be disabled here'}

```

Now track spurious spaces. This is the only part that is automatically checked, using grep in the Makefile.

```
\tracingcommands1
\luadirect{local foo}%
\luaexec{local foo}%
\begin{luacode}
  local foo
\end{luacode}
\begin{luacode*}
  local foo
\end{luacode*}
\tracingcommands0

\end{document}
⟨/testlatex⟩
```