TEX as you like it: The Interpreter package

Paul Isambert

Introduction

This article presents the Interpreter package for LuaTEX, designed to preprocess input files on the fly so that the user can map any syntax to proper TEX and type documents with the language s/he finds more convenient.

This is not a comprehensive description of Interpreter, but only highlights of its functionality; the documentation accompanying the package on CTAN remains the ultimate reference, and contains a complete explanation of an interpretation file.

Motivations

Despite loving TEX, I've always hated typing backslashes and braces for some reason (for one, they're rather badly placed on a French keyboard). At least, the latter can often be avoided thanks to delimited arguments, but unless one is willing to define a new character with catcode 0 (something I find almost counterintuitive) or to venture into the dangerous world of active characters, backslashes cannot be avoided.

Also, I've always found TEX source files (mine and others's) quite unreadable. The likes of `\macro` and `\com{mand}` disturb the normal flow of reading. This became more striking still when I started using the Vim editor. Unlike most text editors, Vim's documentation is made of plain text files meant to be read in the editor itself (this is also true of Emacs); thus one can remain in the same working environment and above all browse the help files as one usually browses some code. If only one could read TEX source files so easily!

All in all, what I wanted was to type TEX source in a syntax unrelated to TEX — a lightweight markup language like Markdown or the syntax used for wikis. Without LuaTEX, the only solution (as far as I know) is to use some script to convert a file into proper TEX (thus creating another file), something I've never tried. With LuaTEX, things change: if you want to preprocess a file on the fly before feeding it to TEX, you can do it, just hook into the `open_read_file` callback!*

Working principles

The basic mechanism behind Interpreter is quite simple; you have a master file in which you input the file(s) to be preprocessed with:

`\interpretfile{`⟨*lang*⟩`}{`⟨*file*⟩`}`

where ⟨*lang*⟩ points to an external file containing the interpretation (explained in the following section). Then Interpreter uses the `open_read_file` callback to control how the lines of ⟨*file*⟩ are to be fed to TEX. This callback is passed a string representing the file to read and should return a table with two entries: `reader`, a function called whenever TEX wants a line, and (optionally) `close`, a function executed when the end of the input file is reached. The simplest implementation of `reader` is to read a line of the input file and return it to TEX; before that, however, one can also modify that line or read others (and perhaps modify them too), which is exactly how Interpreter works. Some practical examples: one can ask Interpreter to change '`some *bold* text`' into '`some \bold{bold} text`' or

```
=========================
=== A section heading ===
=========================
```

into

```
\section{A section heading}
```

or to surround with verbatim macros any material indented with ten spaces, and stop interpreting it at once (so the material is really left verbatim).

Defining simple patterns

As already mentioned, `\interpretfile` will look for an external file matching its first argument; more precisely, if that argument is e.g. `lang`, then the file should be called `i-lang.lua`. It contains all the replacements that will take place to convert the input file; as the extension indicates, the language is Lua. The main function is `interpreter.add_pattern()`, which takes a table defining a pattern to be searched for and replaced with something else. Not surprisingly, one of the entries is `pattern`; another is `replace`; and Interpreter will try to find all material matching the former and replace them with the latter.

For instance, the following will replace `/text/` with `\italic{text}`:†

* This is the reason why Interpreter doesn't work with ConTEXt, in which the callback is frozen. ConTEXt does have modules to process some non-TEX languages, but I'm not aware of a general solution for any language the user might want to define.

† Since the function's single argument is a table, Lua allows the parentheses to be omitted; e.g. `myfn({mytab})` and `myfn{mytab}` are equivalent in such cases. The same is true for strings: `myfn"mystr"` works in the same circumstances.

```
interpreter.add_pattern{
  pattern = "/(.-)/",
  replace = "\\italic{%1}"
}
```

The reader will notice that Lua's magic characters are used, and `(.-)` thus means 'capture the shortest possible sequence made of any number of matching characters', and not a dot followed by a minus sign between parentheses. To denote a magic character itself, one should prefix it with `%`; thus if one wanted to use stars instead of slashes, the pattern should be `%*(.-)%*`, because the star is a magic character (see the Lua reference manual for the list of magic characters). Alternatively, Interpreter has a function `interpreter.nomagic()` which reverses the magic: no character is magic unless prefixed with `%`, except that `...` means the magic `(.-)`. For example, `interpreter.nomagic("*...*")` is equivalent to `"%*(.-)%*"`.

I've mentioned captures, and indeed `replace` makes use of them: `%1` refers to the first (and in this case, only) capture of `pattern`. This follows the behavior of Lua's `string.gsub()`, since ultimately Interpreter uses that function to make the replacement. Accordingly, `replace` can be a string, as is the case here, but also a table (and the entry returned is the one with the first capture, or the entire match if there is no capture, as its key) or a function (to which the captures, or the entire match, are passed as arguments).

Now Interpreter will search all lines for the specified pattern and use the replacement if a match occurs; a limitation (and security) is that matches must be contained in a single line. For instance, the following material will be left untouched:

```
This will /not be
put/ in italics.
```

To span several lines, two solutions are possible. First, one can redefine the pattern to match a single slash, which is converted to `\italics{` or `}` depending on a conditional. To do this, one can use a function in `replace`:

```
local italic
local function makeitalic ()
  if italic then
    italic = false
    return "}"
  else
    italic = true
    return "\\italic{"
  end
end
interpreter.add_pattern{
  pattern = "/",
  replace = makeitalic
}
```

The second solution, sounder and more general, will be explained in the next section.

Before turning to more advanced topics, a word of caution: Interpreter does *not* define TEX macros as `\italic` or `\bold` or `\section`. They are used here because their meaning is clear, but one should obviously use macros defined elsewhere. In other words, Interpreter simply manipulates strings and has nothing to do with typesetting.

Handling paragraphs

Simple patterns are fine as far as they go, but sometimes manipulating input line by line doesn't suffice. For instance, suppose you want to turn

```
1. First item.
2. Second item.
3. Third item.
```

into something like

```
\list
\item First item.
\item Second item.
\item Third item.
\endlist
```

Converting a string of digits followed by a dot at the beginning of a line into `\item` is easy enough. However, how should `\list` and `\endlist` be added to the material?

Such a situation is the reason why Interpreter manipulates paragraphs instead of lines. Instead of fetching a line, converting it according to the defined patterns, and returning it to TEX, Interpreter collects an entire paragraph, does all the conversions, and only then passes it line by line to TEX. In the meantime new lines might have been added.

For Interpreter a paragraph is anything up to and including the first line matching completely the pattern stored in `interpreter.paragraph`, where 'matching completely' means that if the material matching the pattern is removed from the line, the line is empty. By default, `interpreter.paragraph` is defined as `%s*`, i.e. a paragraph is marked by a line containing at most spaces.

To manipulate paragraphs, one should define a pattern with a `call` entry. This should be a function, and it will be executed as follows:

```
function (paragraph, line, index, pattern)
```

The first argument is the entire paragraph where the match occurred. It is represented as a table with nu-

merical indices; `line` is the index of the line where the match occurred, so that `paragraph[line]` returns a string representing that line; `index` is the position in that string where the match was found; finally, `pattern` is the entire table which has been defined with `interpreter.add_pattern`.

Our situation with lists could be solved like this:

```
local item = "^%s*%d+%.%s*"
local function makelist (paragraph)
  for n, l in ipairs(paragraph) do
    paragraph[n] = string.gsub(l, item,
                               "\\item ",1)
  end
  table.insert(paragraph, 1, "\\list")
  table.insert(paragraph, "\\endlist")
end
add_pattern{
  pattern = item,
  call    = makelist
}
```

The following will happen: when Interpreter spots a string of one or more digits followed by a dot at the beginning of a line (spaces notwithstanding), it calls the `makelist` function. This functions searches for the same pattern in all the lines of the paragraph and replaces it with `\item`; also, it inserts new lines with `\list` and `\endlist` at the beginning and the end of the paragraph.

This example used only the first argument of the `call` function. As a more complicated case using all four arguments, let's solve the question of defining `/.../` as a marker for italics possibly spanning several lines. Basically, the solution is identical to the one shown in the previous section: the first slash should be turned into `\italic{` and the second into `}`. But, as already mentioned this solution will be sounder, because the conversion will be done if and only if a pair of slashes is found (so that a slash on its own isn't modified), and also more general, because the same function will be used for all similar patterns.

```
local match,gsub = string.match,string.gsub
local function markup (par, line, index,
                                  pattern)
 local patt = pattern.pattern
 local rep = "\\" .. pattern.replace .. "{"
 if match(par[line], patt, index+1) then
  par[line] = gsub(par[line], patt, rep, 1)
  par[line] = gsub(par[line], patt, "}", 1)
 else
  local n = line+1
  while par[n] do
   if match(par[n], patt) then
    par[line] = gsub(par[line], patt,
                     rep, 1)
    par[n] = gsub(par[n], patt, "}", 1)
    return
   else
    n = n+1
   end
  end
  return index+1
 end
end
interpreter.add_pattern{pattern = "/",
      call = markup, replace = "italic"}
interpreter.add_pattern{pattern = "%*",
      call = markup, replace = "bold"}
interpreter.add_pattern{pattern = '"',
      call = markup, replace = "quote"}
```

Given a pattern, the `markup` function looks for another occurrence of this pattern in the same line or in the following lines of the paragraph. Only if the search succeeds does the replacement happen. Then we specify patterns so `/text/` will be replaced with `\italic{text}`, `*text*` with `\bold{text}`, and finally `"text"` with `\quote{text}`.

Two things should be remarked upon in the code above. First, the line `return index+1` at the end of the function instructs Interpreter to resume its search for patterns at the next position in the current line; without it, the search would start again at the same position where the pattern was found. This `return` statement occurs if no matching character was found, i.e. if the pattern was launched on a lonely slash (or star or double quote). Thus that character was *not* converted, and if the search were to start again at the same position, Interpreter would find the same character, and enter a loop.

Second, the patterns store the macro to be used in the `replace` field. That is totally arbitrary: the table making up the pattern can contain any field. Here the `replace` entry can be used because if a pattern has both `call` and `replace`, the latter is ignored (i.e. the mechanism described in the previous section doesn't apply).

Bells and whistles

As said in the introduction, this paper is not a complete manual for Interpreter. Here I'll mention a few other bits of functionality.

First and foremost, the search for patterns is done according to an order. Each pattern belongs to a class, as specified by the `class` entry in the pattern table (this entry defaults to the number recorded in

interpreter.default_class), and classes are applied one after the other in ascending order; patterns belonging to the same class are ordered by length and are applied from longer to shorter.

One of the reasons why classes are important is so input can be protected, i.e. prevent Interpreter from converting some lines or an entire paragraph. For instance, consider a pattern denoting verbatim material. It will launch a `call` function to add something like `\verbatim` and `\endverbatim` as the first and last lines. In addition, it should call the function `interpreter.protect()`, to stop Interpreter from manipulating the current paragraph, i.e. other patterns won't be searched for and replaced, as expected for verbatim material. Such a pattern should belong to the very first class, so that it is executed before all the others; otherwise, protection would be only partial.

Another way to protect input, this time locally, is to record a pair of strings as left and right markers such that the enclosed material shouldn't be touched. The function `interpreter.protector()` does this; e.g. after `interpreter.protector('"')`, material between double quotes will be left intact (if the function is called with only one argument, it is used for both the left and right markers).

Interpreter allows you to mix different syntaxes, or rather, it has no notion of well-formedness for the language you define. Thus usual TEX commands can be used in the middle of an interpreted file. One convenient trick is to define an easy syntax to add new patterns to the file being read. For instance, with

```
interpreter.add_pattern{
  class   = 1,
  pattern = "^DEF%s*(.-)%s*=%s*(.+)",
  replace = function (pat, rep)
      interpreter.add_pattern{
        pattern = pat,
        replace = rep}
        return ""
      end
}
```

simple new patterns can be created as follows:

```
DEF pattern = replacement text
```

I let the reader check that it works properly.

Input files that look like main files

One of Interpreter's limitations is that it works only on input files: it can't work with a main file directly fed to TEX. The reason for this is that it uses the `open_read_file` callback which, as its name implies, concerns `\input` and `\read` (`\interpretfile` ultimately boils down to `\input`).

The main file could be manipulated with the `process_input_buffer` callback, but this isn't as flexible as `open_read_file`, and most importantly it doesn't attach to a specific file. Yet one can have the impression to work on the main file by invoking LuaTEX as follows (this works for plain TEX; LATEX users should adapt accordingly):

```
luatex -jobname=⟨file⟩
       \input interpreter
       \interpretfile{⟨language⟩}{⟨file⟩}
       \bye
```

The important point is to set `-jobname` to the input filename, so the relevant output files (the PDF, log, etc.) are created with the proper name. It might be wise to set `-output-directory` to the file's directory too, but that is not necessary.

Of course, one can also input other files besides Interpreter. It might also be interesting to use a Lua initialization script, an alternative I won't investigate here.

Conclusion

LuaTEX keeps changing my TEX world every day: new horizons, new solutions, a new language. With Interpreter, even my usual TEX source doesn't look the same! I'm even working on a document where unmarked macros represent themselves, i.e. `\macro` is turned to `\string\macro`.

One thing I do not know is whether Interpreter would be convenient to process something like XML; not using that language, I haven't tried to create an interpretation file for it, and I wonder whether Interpreter would be up to the task or would rather get in the way. If the reader finds a solution, just let me know!

◇ Paul Isambert
zappathustra (at) free dot fr

Postscript: Just before this article went to press, Interpreter was rewritten with the Gates package. None of what is said here needs updating, since Interpreter hasn't changed on the surface, but its implementation now allows deep hacking, because it is made of small logical steps that can be externally controlled. Details can be found in the new version's documentation and general principles in the documentation for Gates.