Interpreter

v.1.2, June 2012

Paul Isambert
zappathustra AT free DOT fr

Introduction

Interpreter preprocesses input files before their contents is fed to TEX. It is meant to write document with whatever markup one wishes to define while using normal TEX macros in the background. As a simple example, suppose you have a macro `\bold` to put text in boldface ; then *Interpreter* lets you map `*text*`, or `<strong>text</strong>`, or simply `!text`, or anything else, to `\bold{text}`. *Interpreter* doesn't perform any trickery with active characters ; instead, it manipulates the strings representing the lines of a file and search for patterns.

There are two main advantages : first, TEX documents can be typeset with a completely non-TEX syntax ; second, if one uses some lightweight markup language, the source file is much easier to read and might even be more useful than the typeset PDF file, e.g. for some technical documentation you want to read directly in your text editor while writing code (powerful editors generally have their own documentation in such a format, for a good reason). A third advantage, not explored in this documentation, is that while feeding modified lines to TEX you can also translate the original lines into, say, HTML, and write them to an external file, thus creating both PDF and HTML output at once.

Interpreter has been rewritten with the Gates package (actually, only the Lua side) in version 1.1. That hasn't changed anything to its default behavior, but now it can also be customized quite deeply, since its code is a collection of small chunks with names that can be externally controlled and/or augmented. See the Gates documentation for further information. The last sections of this documentation describe the gates in *Interpreter*.

Input files

Once *Interpreter* is loaded with

```
\input interpreter
```

in plain TEX or

```
\usepackage{interpreter}
```

in LaTEX, files to be processed are input as follows :

```
\interpretfile{<language>}{<file>}
```

There should exist a file `i-<language>.lua` containing the language used in ‹`file`›. For instance, the source of this documentation is `interpreter-doc.txt`, input in the master file `interpreter-doc.tex` with

```
\interpretfile{doc}{interpreter-doc.txt}
```

and the interpretation to be used is defined in `i-doc.lua`. The contents of such an interpretation file is the object of the rest of this documentation.

Paragraphs

Interpreter doesn't process lines one by one. Instead, it gathers an entire paragraph and then processes the lines. It is important because you can manipulate an entire paragraph when a given pattern is detected, and modify several lines according to what happens in only one. A paragraph in *Interpreter* has nothing to do with what TEX considers a paragraph ; instead, it is defined by the following string.

`interpreter.paragraph` *(Default : blank line with spaces ignored)*
A string to be interpreted as a paragraph boundary when *Interpreter* collects lines before processing them. The string actually represents a pattern, so magic characters are obeyed. The default is `%s*`, i.e. a blank line is considered a paragraph boundary, spaces notwithstanding. Of course, the end of the file itself is a paragraph boundary.

Declaring patterns

Once the lines of a paragraph have been collected, *Interpreter* searches them trying to match declared patterns, but it doesn't do so indiscriminately : patterns are searched in a given order, as explained below.

Patterns are searched for in each line only, i.e. no match can occur across lines. However, since you can manipulate entire paragraphs based on a match in one line, the limitation easily vanishes.

`interpreter.add_pattern(‹table›)`
This is the basic function used to defined patterns. The `‹table›` may contain the following entries, along other entries *Interpreter* won't use but which can be useful to you, especially with `call` below. The function returns a table.

`class` *(Default: `intepreter.default_class`)*
The class of the pattern. See the section on classes.

`pattern`
The pattern to match. Lua's magic characters are in force and should be escaped with `%` if necessary, unless `nomagic` is `true` (or the pattern itself is the result of `interpreter.nomagic`).

`nomagic` *(Default: `false`)*
A boolean deciding whether the pattern should be transformed with `interpreter.nomagic`.

`replace`
The replacement for the pattern, applied only if there is no `call` entry. This may be a string, a table or a function. *Interpreter* simply executes something similar to `string.gsub()`, hence the replacement follows this function's ordinary syntax. More precisely, if `replace` is a string, the pattern is replaced with it; in this string, `%n` may be used to denote the *n*th capture in the pattern. If `replace` is a table, the first capture or the entire match (if there is no capture) is used as the key, and the associated value is used as the replacement. If `replace` is a function, it is called with the captures passed as arguments, or the entire match if there is no capture. For instance, the following pattern will replace all `*text*` with `\bold{text}`:

```
interpreter.add_pattern{
  pattern = "%*(.-)%*",
  replace = [[\bold{%1}]]
}
```

`offset` *(Default : 0)*

The number of positions *Interpreter* should shift to the right after a match has occurred. Normally, *Interpreter* starts searching for another occurrence of the current pattern at the same position where it found the last one. However, loops might easily occur : the replacement for a pattern may very well contain another match for the same pattern, so *Interpreter* will get stuck. Suppose for instance you want to replace `TeX` with `\TeX`. The first match will do that, but then *Interpreter* will start searching again at the backslash, producing `\\TeX`, then `\\\TeX`, etc. In this case, if you set `offset` to 2 in the pattern, then search will start again at the `e` and no new match will occur.

`call`

This entry shall contain a function to be called if there is a match (if this entry exists, `replace` isn't applied). It is meant to perform complex tasks that aren't amenable to simple string replacement. The function will be executed as follows :

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

`paragraph` is a table representing the current paragraph ; lines are stored at successive indices. The last line of this paragraph is always the paragraph boundary (see `interpreter.paragraph`), unless the paragraph stopped at the end of the file. The second argument, `line`, is a number representing the index in `paragraph` containing the line where the pattern was found ; `index` is the position in this line where the match occurred. Finally, `pattern` is the entire table declared with `interpreter.add_pattern` and containing all the entries discussed here.

The function may return zero, one, or two numbers. If it returns none, the search for the next occurrence of the pattern will start again on the same line (rather, on the line with the same position in the paragraph), at `index`. If it returns one number, the search will resume at the same line but at position *n*, with *n* the returned number. Finally, if two numbers are returned, the search will resume at line *m* at position *n*, *m* and *n* being the returned values. Specifying which line should be examined when the search resumes might be necessary if the function adds new lines in the paragraph *before* the current line, since *Interpreter* only keeps count of line numbers.

The entire paragraph can thus be modified if necessary. For instance, suppose you want to declare comments in your source file with only `!Comment` in the first line, i.e. TeX should ignore a paragraph such as :

```
!Comment
This should be ignored
by TeX
```

Then the following pattern will do (where the function requires only the first argument) :

```
local function comment (paragraph)
  for n, l in ipairs(paragraph) do
    paragraph[n] = "%" .. l
  end
end
interpreter.add_pattern{
  pattern = "^!Comment",
  call    = comment
}
```

`interpreter.nomagic (string)`

A function which reverses the usual Lua magic for patterns : ordinary magic characters are normal characters here, unless they are prefixed with `%`, in which case they are magic again. For instance, a pattern like `.+` is normally interpreted as "one or more characters". If passed to this function, a pattern is returned meaning "a dot followed by a plus sign". On the contrary, `%.%+` normally has the second interpretation, while with `interpreter.nomagic` it has the first one. The function makes another transformation : `...` is used to denote a capture `(.-)`. Thus `interpreter.nomagic('*...*')` returns a pattern matching any number of characters surrounded by stars and capturing those characters ; this would be expressed in ordinary Lua magic as `%*(.-)%*`.

Classes

As already alluded to, the search for patterns isn't done at random. Instead, patterns are organized in classes, which are applied one after the other. More precisely, the process is as follows : *Interpreter* searches the entire paragraph for the first pattern in class 1, then for the second pattern in the same class, then for the third, etc., then when there is no pattern left in class 1 it does the same with class 2, up to class *n*, where *n* is the highest class number such that there exists a class *n - 1* (in other words, classes should be numbered consecutively). Finally, the same goes for the patterns in class 0 (which always exists, even if it contains no pattern).

Inside a class, patterns are ordered by length from long to short, or alphabetically if two patterns have the same length. This means that if you use e.g. `/text/` for italics and `//text//` for bold, you don't need to put the second pattern in a class before the first to avoid `//text//` being interpreted as two empty arguments in italics surrounding a text in roman. Since the way the bold-pattern will be declared, e.g. `//(.-)//`, is probably longer than for the italic-pattern, e.g. `/(.-)/`, it will always match first.

That said, the sorting isn't very clever and simply relies on the number of symbols, no matter what they mean ; in the patterns above, the parentheses denote a capture but they still count in the pattern's length as understood by *Interpreter*. Alternatively, while `.*` denotes "zero or more character" and `%+` means "a plus sign" (`+` being magic, you have to escape it to refer to it), in *Interpreter*'s eye the two patterns have the same length : two. Finally, one should be aware that patterns declared with a `nomagic` entry set to `true` are sorted after they've been transformed (so that their real length might not be obvious). So classes are needed when patterns need a proper ordering no matter their lengths. For instance, some patterns should always be declared first, as they protect input from *Interpreter* (see next section), while others might need to be declared last, as they rely on what previous patterns might have done. Besides, classes are metatables for the patterns they contain.

`interpreter.default_class` *(Default : 1)*

All patterns belong to a class, even though you may omit the `class` entry when declaring one. In this case, the pattern is assigned to the class denoted by this number.

`interpreter.set_class(number, table)`

Defines class `number` as `table`. Classes don't need to be defined beforehand for patterns to be added to them (rather, *Interpreter* defines them implicitly when needed). However, classes are also metatables for the patterns, so that if there lacks an entry in a pattern's table, the class's entry is used if it exists. The function returns a table.

Protecting input

Sometimes you want *Interpreter* to refrain from interpreting ; that is most useful for verbatim code, for instance. There are various ways to do that.

`interpreter.active` *(Default : true)*

A boolean switching *Interpreter* on and off. Beware, the switching applies only starting at the next paragraph.

`interpreter.protect([line])`

A function protecting all or part of the current paragraph. If `line` is given, it should be a number *n*, and line *n* in the current paragraph will be protected ; without `line`, the entire paragraph is protected. Protecting means that the patterns not yet searched for will be ignored. For instance, if you want material to be read verbatim when surrounded with `<code>` and `</code>`, you can declare a pattern as follows :

```
local function verbatim (buffer)
  buffer[1] = "\\verbatim"
  buffer[#buffer - 1] = "\\endverbatim"
  intepreter.protect()
end
interpreter.add_pattern{
  pattern = "^%s*<code>%*s$",
  call     = verbatim,
  class    = 1
}
```

This code is extremely simplified: it assumes that `<code>` and `</code>` starts and ends the paragraph and that `</code>` isn't the last line of the file (otherwise it'd also be the last line in the paragraph, whereas here the last one is the paragraph boundary). An important point is that the pattern belongs to the first class, so it is called before all other patterns (provided there is no shorter pattern in class 1) and prevents them from doing anything, since the entire paragraph is protected. (Typesetting the material as verbatim material obviously depends on the `\verbatim` macro, not on *Interpreter*.)

`interpreter.escape`

A character which prevents patterns from being replaced if immediately preceded by it. As an example, if `interpreter.escape = '_'`, and `*text*` denotes italic, then `*text*` will produce *text* while `_*text*` will produce *text*. Once a paragraph has been processed, *Interpreter* removes all escape characters. Only one character can be an escape character.

`interpreter.protector(left[, right])` *(right defaults to `left`)*

Defines two characters to protect what they surround. In other words, *Interpreter* replaces patterns only if the match isn't found between `left` and `right`. Unlike the escape character, you can define as many protectors as you wish; and unlike the escape character again, *Interpreter doesn't* remove them once the paragraph has been processed, so you must take care of them. For instance:

```
intepreter.protector('"')
interpreter.add_pattern{
  pattern = '"(.-)"',
  replace = '\\verb`%1`',
  class   = 0
}
```

Anything between double quotes will be left untouched ; then, when the paragraph has been processed for all other classes, a pattern in class `O` calls the `\verb` command to take care of the argument. Note that the protectors should enclose what they protect without coinciding with it ; this is not the case here, which is why the pattern is applied.

`interpreter.direct`
(Default : two percent signs then `I` and at least one space)
A string, actually a pattern, signalling that the line which it begins should be processed as Lua code. The default is `%%%%I%s+`, i.e. `%%I` followed by at least one space. The pattern shouldn't declare itself as attached to the beginning of the line (as in `^%%%%I%s+`) because they will be matched at the beginning of the line only anyway. The line is processed with the `loadstring` function, and then turned into an empty line. For instance :

```
%%I interpreter.active = false
This won't be interpreted...
%%I interpreter.active = true
```

As this example shows, lines flagged with `interpreter.direct` don't obey `interpreter.active` and are always processed as described above.

Technical stuff
You don't have to bother with this section if you don't mind how *Interpreter* does its job ; actually you won't learn much anyway.

`interpreter.reset()`
A function which resets everything to default and deletes classes. It is used when calling `\interpretefile` so that new interpretetions start from zero.

`interpreter.register(function)`
A function called to put *Interpreter*'s main function into the `post_linebreak_filter` callback ; you can redefine it at will. If it is undefined, `callback.register()` is used, unless `luatexbase.add_to_callback()` is

detected. (The detection takes place at the first call to `\interpretfile`, so there is no need to load *Interpreter* after `luatexbase`.)

```
interpreter.unregister(function)
```

A function called to remove *Interpreter*'s main function from the `post_linebreak_filter` callback. It works similarly to the previous one.

An example : i-doc.lua

Here's a description of `i-doc.lua`, the file containing the interpretation used for *Interpreter*'s documentation. Remember that none of the TEX macros used here is defined by *Interpreter* ; instead, they are my own and should be adapted if necessary. Also several options taken here are far from optimal but are convenient examples.

Shorthands for often used functions.

```
local gsub, match = string.gsub, string.match
local add_pattern = interpreter.add_pattern
local nomagic     = interpreter.nomagic
```

Class 1 and 2 will be used for verbatim (thus protecting) and "normal" patterns go into class 3 or higher.

```
interpreter.default_class = 3
```

The reader might have observed that `interpreter-doc.txt` begins with a table of contents. This table is useful for the source file only, and isn't typeset by TEX, because the following pattern suppresses it : the entire paragraph containing `TABLE OF CONTENTS` on a line of its own is deleted. Protecting the paragraph is useless, but it makes things a little bit faster because the paragraph won't be pointlessly searched for other patterns.

```
local function contents (buffer)
  for n in ipairs(buffer) do
    buffer[n] = ""
  end
  interpreter.protect()
end
add_pattern{
  pattern = "^%s*TABLE OF CONTENTS%s*$",
  call    = contents,
  class   = 1
}
```

Sections headers are typeset as

```
====================================== section_tag
=== Section title ====================
======================================
```

The first and third line are decorations and they are removed. The `section_tag` is meant for the source only again (linking the section to the table of contents). I could have used it to create PDF destinations, but that seemed unnecessary in such a small file. The associated pattern is : at least four equals signs.

```
add_pattern{
  pattern = "^====+.*",
  replace = ""
}
```

The middle line is spotted with the tree equals sign at the beginning of the line (the previous pattern being longer, the decoration lines have been already removed and they won't be taken for section titles). The signs are removed and replaced with `\section{` and `}`.

```
local function section (buffer, num)
  local l = buffer[num]
  l = gsub(l, "^===%s*", "\\section{")
  l = gsub(l, "%s*=+%s*", "}")
  buffer[num] = l
```

```
end
add_pattern{
  pattern = "^===",
  call    = section
}
```

The following pattern simply turns `Interpreter` into `\ital{Interpreter}`. The meaning of the `\ital` command is obvious, I suppose. Note the offset : starting at the backslash, this leads to the *n* in *Interpreter*, thus avoiding matching the pattern again. The Lua notation with double square brackets is used for strings with no escape character (hence `\ital` and not `\\ital` as would be necessary with a simple string).

```
add_pattern{
  pattern = "Interpreter",
  replace = [[\ital{Interpreter}]],
  offset  = 7
}
```

Turning `TeX` into TEX. This illustrates the use of a function as `replace` ; the point is that `\TeX` should be suffixed with a space if initially followed by anything but a space or end of line (so as not to form a control sequence with the following letters), and it should be suffixed with a control space if initially followed by a space or end of line (so as to avoid gobbling the space). So the function checks the second capture. Note that simply replacing `TeX` with `\TeX{}` would be much simpler, but less instructive !

```
local function maketex (tex, next)
  if next == " " or next == "" then
    return [[\TeX\ ]]
  else
    return [[\TeX ]] .. next
  end
end
add_pattern{
  pattern = "(TeX)(.?)",
  replace = maketex,
  offset  = 2
}
```

The following turns `<text>` into ‹`text`› and `_text_` into *text*. Setting a class just so the patterns inherit the `nomagic` feature is of course an overkill, but that’s an example.

```
interpreter.set_class(4, {nomagic = true})
add_pattern{
  pattern = "<...>",
  replace = [[\arg{%1}]],
  class   = 4
}
add_pattern{
  pattern = "_..._",
  replace = [[\ital{%1}]],
  class   = 4
}
```

I use double quotes as protectors ; they are replaced with a `\verb` command at the very end of the processing (with class 0).

```
interpreter.protector('"')
add_pattern{
  pattern = nomagic'"..."',
  replace = [[\verb`%1`]],
  class   = 0
}
```

The description of functions (in red in the PDF file) are handled with the `\describe` macro, which takes the function as its first argument and additional information as its second one (typeset in italics in the PDF file). In the source, it is simply marked as

```
> function (arguments) [Additional information]
```

with `[Additional information]` sometimes missing (i.e. there is no empty pairs of square brackets). Descriptions of entries in pattern tables follows the same syntax, except the line begins with >>. So the pattern first spots lines beginning with `>[>]` followed by at least one space, adds an empty pair of brackets at the end if there isn't any, and turn the whole into `\describe`. The number of > symbols sets `\describe`'s third argument, which specifies the level of the bookmark.

```
local function describe (buffer, num)
  local l = buffer[num]
  if not match (l, "%[.-%]%s*$") then
    l = l .. " []"
  end
  local le = match(l, ">>") and 4 or 3
  buffer[num] = gsub(l, ">+%s+(.-)%s+%[(.-)%]",
               [[\describe{%1}{%2}{]] .. le .. "}")
end
add_pattern{
  pattern = "^>+%s+",
  call    = describe
}
```

Here's how multiline verbatim is handled ; in the source it is simply marked by indenting the line with ten spaces ; thus code is easily spotted when reading the source without useless and annoying `<code>/</code>` or anything similar to mark it. To be properly processed by TEX, the code should be surrounded by `\verbatim` and `\verbatim/` (my way of signalling blocks). Those must be on their own lines, so we insert a line at the beginning and at the end of the paragraph : for the closing `\verbatim/`, we can simply replace the last line of the paragraph, which is the boundary line, unless we're at the end of the file. But for the opening `\verbatim` a line must be added at the beginning of the paragraph ; thus line numbers in the original source file and in its processed version don't match anymore, and this might be annoying when TEX reports erros. Besides, blank verbatim lines aren't handled correctly and create a new verbatim block instead. So this way of marking verbatim material is good for small documents, but explicit marking is cleaner and more powerful (albeit not so good-looking in the source file).

Note that the verbatim pattern belongs to class 2 and the entire paragraph is protected, so *Interpreter* leaves it alone afterward (remember the default class is 3). Of course, the first ten space characters are removed.

```
local function verbatim (buffer)
  for n, l in ipairs(buffer) do
    buffer[n] = gsub(l, "%s%s%s%s%s%s%s%s%s%s","", 1)
  end
  table.insert(buffer, 1, [[\verbatim]])
  if gsub(buffer[#buffer],
          interpreter.paragraph, "") == "" then
    buffer[#buffer] = [[\verbatim/]]
  else
    table.insert(buffer, [[\verbatim/]])
  end
  interpreter.protect()
end
add_pattern{
  pattern = "^%s%s%s%s%s%s%s%s%s%s",
  call    = verbatim,
  class   = 2
}
```

And now comes the fun part. I wanted `i-doc.lua` to be self-describing. The source of what you're reading right now isn't `interpreter-doc.txt`, but `i-doc.lua` itself input in the latter file with

```
\intepreterfile{doc}{i-doc.lua}
```

How should code and comment be organized in `i-doc.lua`? Well, there is little choice, since the file is a normal Lua file : comment lines should be prefixed with `--` or surrounded with `--[[` and `--]]`. I chose the latter option, which is simpler. But normal code should also be typeset as verbatim material ; I could have begun all lines with ten spaces, but that would have seemed strange. Instead, `--]]` is turned into `\source` and `\source/` is added at the end of the paragraph (`\source` is just `\verbatim` with a different layout). Which means all paragraphs have the same structure : comments between `--[[` and `--]]` and code immediately following (`--[[` is simply removed). The pattern is in class 1 and the paragraph is protected, so that lines indented with ten spaces or more aren't touched by the previous verbatim pattern (in class 2).

```
local function autoverbatim (buffer, line)
  buffer[line] = [[\source]]
  for n = line + 1, #buffer do
    interpreter.protect(n)
  end
  if gsub(buffer[#buffer],
          interpreter.paragraph, "") == "" then
    buffer[#buffer] = [[\source/]]
  else
    table.insert(buffer, [[\source/]])
  end
end
add_pattern{
  pattern = nomagic"%^--]]",
  call    = autoverbatim,
  class   = 1
}
local function remove_comment ()
  return ""
end
add_pattern{
  pattern = nomagic"%^--[[",
  replace = remove_comment
}
```

The Gates in *Interpreter*

Interpreter is written with the Gates package (only the Lua side, actually). It means that it can be hacked down to the core. Here I'll simply list the gates involved ; you should read the Gates documentation to learn how to use them.

There are three gates families : `interpreter`, associated with the main `interpreter` table, contains the user interface ; `interpreter_tools`, associated with `interpreter.core.tools` table, contains internal functions ; finally `interpreter_reader`, associated with the `interpreter.core.reader` table, contains the main functions used to read the file.

Whenever I mention a conditional or a loop, I mean the local conditionals and loops, relative to the l-gate where the gate appears. Also, the syntax indicates the arguments a gate uses, not all the arguments that are passed to it (which are simply what the previous gate has returned).

As an example of customizing *Interpreter* with Gates, you could very well add a bit of code which does something to all lines. Inserting a small gate, say `everyline`, after `check_direct` in `aggregate_lines` below would do the trick, e.g.:

```
function interpreter.core.reader.everyline (file, line)
  line = dosomething(line)
  return file, line
end
interpreter.core.reader.add(
  "everyline", "aggregate_lines", "after check_direct")
interpreter.core.reader.conditional(
  "everyline", "aggregate_lines", function (f, l) return l end)
```

(Note that it is important to check that the line really exist, because one might have hit the end of the file ; hence the conditional, as with others gates in `aggregate_lines`).

The interpreter table

All the user functions in `interpreter` are simple m-gates, so they can be treated as ordinary functions, except `interpreter.add_pattern`, which is an l-gate containing, built as :

```
add_pattern
. ensure_class
. apply_nomagic
. insert-pattern
. . do_insert
. . sort_class
```

`ensure_class (‹pattern›)` *(m-gate)*
Creates the class of ‹`pattern`› if necessary, and set it as the metatable for ‹`pattern`›. Classes themselves are kept in the `interpreter.core.classes` table. The gate return ‹`pattern`› and the class number.

`apply_nomagic (‹pattern›, ‹class›)` *(m-gate)*
Transforms the `pattern` entry in ‹`pattern`› with `intepreter.nomagic`; tied to a conditional that returns ‹`pattern`›'s `nomagic` entry (so the gate is executed only if `nomagic` is true) ; autoreturns both arguments.

`insert_pattern (‹pattern›, ‹class›)` *(l-gate)*
An autoreturning l-gate containing the following two gates.

`do_insert (‹pattern›, ‹class›)` *(m-gate)*
Adds ‹`pattern`› to ‹`class`› (i.e. `interpreter.core.classes[<class>]`).

`sort_class (‹pattern›, ‹class›)` *(m-gate)*
Sorts ‹`class`› with function `interpreter.core.tools.sort`. This gate can be skipped to apply the patterns in the order in which they were declared.

The interpreter.core.tools table

All the functions in the `interpreter.core.tools` all are simple m-gates.

`sort (‹patt1›, ‹patt2›)` *(m-gate)*
Returns true if the pattern in `patt1` is longer than the one in `patt2`, or if they have the same length and the first ranks before the second with respect to alphabetical order. The gate is used in the `interpreter.sort_class` m-gate.

`xsub (‹string›, ‹index›, ‹pattern›, ‹replacement›)` *(m-gate)*
Returns `‹string›` with `‹pattern›` replaced with `‹replacement›`, but only once, and only after `‹index›`.

`protector (‹string›, ‹index›)` *(m-gate)*
Checks whether `‹index›` in `‹string›` isn't between characters declared with `interpreter.protector`. If that is the case, the function returns `nil` and the index of the second protector. Otherwise, it returns `‹index›`.

`get_index (‹string›, ‹pattern›, ‹index›)` *(m-gate)*
Checks whether `‹pattern›` occurs in `‹string›`, starting at `‹index›`. If it does, but if `‹index›-1` is `interpreter.escape`, calls itself with `‹index›+1`. Otherwise, calls `interpreter.core.tools.protector` to check whether `‹index›` is in a protected part of the string. If so, calls itself with `‹right›+1` instead of `‹index›`, where `‹right›` is the second return value of of `interpreter.core.tools.protector`, i.e. it searches again after the right protector. If `‹index›` is found, end of story, returns `‹index›`, otherwise returns nothing.

The `interpreter.core.tools` table also contains `magic_characters`, a table with an entry for each magic character in Lua except '`.`' and '%' ; the values to those entries are the same characters prefixed with '%'. The table is used by `interpreter.nomagic` to spot and replace magic characters, with the dot and the percent sign dealt with independantly.

The interpreter.core.reader table

Interpreter works by hooking in the `open_read_file` callback ; the function registered there is the `interpreter.core.reader.input` l-gate, built as follows :

```
input
. unregister
. . set_unregister
. . use_unregister
. open_file
. set_reader
```

`unregister (‹filename›)` *(l-gate)*
Contains the following two m-gates; `‹filename›` is received from `input`, which itself receives it from the callback, i.e. that's the file that's being input (the second argument to `\interpretfile`). It is also automatically returned.

`set_unregister ()` *(m-gate)*
Sets the function to remove `input` from the callback, namely `interpreter.unregister`; the gate is called only if gate `interpreter.unregister` doesn't already exits. If `luatexbase` is detected, the functions there are used; otherwise, `callback.register` is used with `nil` as the second argument.

`use_unregister ()` *(m-gate)*
Calls `interpreter.unregister()`. (You don't want the next input file to be processed with *Interpreter* by default, that's why you remove the callback function; not that the current one is nonetheless processed with the current file, of course.)

`open_file (‹filename›)` *(m-gate)*
Returns `io.open(<filename>)`.

`set_reader (‹file›)` *(m-gate)*
Returns a table with a `reader` entry containing a function whose definition is

```
function ()
  return interpreter.core.reader.read_file(f)
end
```

That's the convention for the `open_read_file` callback: it should return such a table, and the function will be called each time a line is required from the input file.

So most of the work is done by `interpreter.core.reader.read_file`, which is why it is so heavy; it receives a file handle:

```
read_file
. make_paragraph
. . aggregate_lines
. . . read_line
. . . check_direct
. . . insert_line
. . apply_classes
. . . pass_class
. . . . pass_pattern
. . . . . process_lines
. . . . . . switch
. . . . . . call
. . . . . . replace
. . . . . . protect
. . . unprotect
. . . . undo_protected
. . . . unprotect_lines
. . . remove_escape
. return_line
```

`make_paragraph (‹file›)` *(l-gate)*
The big l-gate that contains everything that follows, barring `return_line`. It is called if and only if the `interpreter.core.lines` table is empty; that table is where lines of a paragraph are stored, and it is emptied by `return_line`.

`aggregate_lines (‹file›, ‹line›)` *(l-gate)*
The main l-gate that reads line from `‹file›` and stores them in `interpreter.core.lines`. It loops until `‹line›` is `nil` or equivalent to `interpreter.paragraph`. (Of course, `‹line›` is `nil` on the first iteration, but the `loopuntil` conditional is evaluated after that first iteration, during which the last subgate `insert_line` will probably returns a line.)

`read_line (‹file›)` *(m-gate)*
Reads the next line from `‹file›`, and returns `‹file›` and that line.

`check_direct (‹file›, ‹line›)` *(m-gate)*
If `‹line›` begins with `interpreter.direct`, removes it and use `loadstring` on the resulting string. Returns `‹file›` and `‹line›`, the latter set to an empty string is the previous operation applied. The gate depends on a conditional : `‹line›` should be non-`nil` (of course), and `interpreter.direct` should be defined.

`insert_line (‹file›, ‹line›)` *(m-gate)*
Adds `‹line›` to `interpreter.core.lines`. Automatically returns the two arguments (and if `‹line›` isn't `nil` or equivalent to `interpreter.paragraph`, it will be executed again).

`apply_classes ()` *(l-gate)*
The l-gate that applies transformations to the lines, once the paragraph has been gathered, with the gates that follow. For each class, it will apply each pattern on each line. It depends on a conditional : `interpreter.core.lines` shouldn't be empty, and `interpreter.active` should be true.

`pass_class ()` *(l-gate)*
This gate iterates on all classes in `interpreter.core.classes` and then on class 0. On each iteration it checks beforehand whether the paragraph is protected, i.e. `interpreter.core.reader.protected` isn't a boolean (see `unprotecte` below). On each iteration, the class number and the class itself are returned. (This behavior is implemented with a Gates iterator.)

`pass_pattern (‹ignored›, ‹class›)` *(l-gate)*
Same as `pass_class`, except it iterates on the patterns in `class` : it is executed as long as `interpreter.core.reader.protected` and returns the patter number and the pattern itself. (The `‹ignored›` argument isn't used ; it is for the `pass_class` iterator ; the same holds for the following gates.)

`process_lines (‹ignored›, ‹pattern›)` *(l-gate)*
Again, this calls an iterator. It browses each line in `interpreter.core.lines` and returns the line's number (provided it is valid, i.e. not a table, see `protect` below), ‹`pattern`› and the current index in that line. To keep track of the current line and index, two internal numbers are used: `interpreter.core.reader.current_line` and `interpreter.core.reader.current_index`.

`switch (‹line›, ‹pattern›, ‹index›)` *(m-gate)*
If ‹`pattern`› has a `call` entry, it sets the `call` gate below to `ajar`; otherwise, if ‹`pattern`› has a `replace` entry, it sets the `replace` gate to `ajar`.

`call (‹line›, ‹pattern›, ‹index›)` *(m-gate)*
This gate is closed by default and set to `ajar` by `switch` above. If, starting at ‹`index`›, the ‹`pattern`›'s `pattern` entry can be found in ‹`line`› with `interpreter.core.tools.get_index` (which makes sure that protectors are obeyed and returns ‹`newindex`›, where the pattern is found if it occurs), the ‹`pattern`›'s `call` entry is applied as

```
<pattern>.call(interpreter.core.lines,
               <line>, <newindex>, <pattern>)
```

This may returned zero, one or two values. If nothing is returned, `interpreter.core.reader.current_index` is set to 0, which makes the `process_lines` iterator consider the next line. If one value is returned, it is the new current index and `process_lines` will not update the line number. If two values are returned, the first is the new current line number and the second the new current index.

`replace (‹line›, ‹pattern›, ‹index›)` *(m-gate)*
This gate is closed by default and set to `ajar` by `switch` above. This tries to find the ‹`pattern`›'s `pattern` like `call` above, and if it is found, it applies `interpreter.core.tools.xsub` as:

```
xsub (interpreter.core.lines[<line>],
      <newindex>, <pattern>.pattern, <pattern>.replace)
```

where ‹`newindex`› is defined as in `call`. The return value of `xsub` is assigned to `interpreter.core.lines[<line>]`, and the current index is set to ‹`index`› plus the ‹`pattern`›'s `offset` if any. If the pattern wasn't found, the current index is set to 0, which makes `process_lines` turn to the next line as explained in `call`.

`protect ()` *(m-gate)*

The `interpreter.protect()` function can either protect the whole paragraph (when no argument is passed) or a single line (when a number is passed). In the first case, `interpreter.core.tools.protected` takes the value `true`, which is checked in various gates above. In the second case, `interpreter.core.tools.protected` is a table with each index indicating a line to be protected. This gate implements the protection in that case: it iterates on all entries in the table with `pairs` and protects the line with the same index in `interpreter.core.lines` by transforming it into a table (with a single entry, the string representing the original line); the type of the line is checked in the `process_lines` iterator above. The gate's iterator doesn't take arguments, but the function itself is defined as taking a number (the line).

`unprotect ()` *(l-gate)*

Now all the patterns in all the classes have been applied to the entire paragraph, and protection must be removed. This l-gate contains the following two gates.

`undo_protected ()` *(m-gate)*

Simply sets `interpreter.core.tools.protected` to `nil` so it is ready for the next paragraph.

`unprotect_lines ()` *(m-gate)*

Restores all the lines in `interpreter.core.lines` as simple strings. The gate uses an iterator that simply runs `ipairs` on `interpreter.core.lines`, so the function's definition actually takes the line's number and the line itself as arguments.

`remove_escape ()` *(m-gate)*
If `interpreter.escape` is defined, removes all its occurrences in each line of the paragraph.

This is the end of the big `make_paragraph` l-gate. It won't be called again until the paragraph has been fully passed to TEX, i.e. when `interpreter.core.lines` is empty.

`return_line ()` *(m-gate)*
Pops the first line from `interpreter.core.lines` and returns it. Since this is the very last subgate of `read_file`, the line is passed to TEX.

Typeset with LuaTEX 0.71 in Chaparral Pro and Lucida Console … nonetheless this documentation looks dull, I don't know why.