LPeg.re

Regex syntax for LPEG

The `re` Module

The `re` module (provided by file `re.lua` in the distribution) supports a somewhat conventional regex syntax for pattern usage within **LPeg**.

The next table summarizes `re`'s syntax. A `p` represents an arbitrary pattern; `num` represents a number (`[0-9]+`); `name` represents an identifier (`[a-zA-Z][a-zA-Z0-9_]*`). Constructions are listed in order of decreasing precedence.

Syntax	**Description**
`( p )`	grouping
`& p`	and predicate
`! p`	not predicate
`p1 p2`	concatenation
`p1 / p2`	ordered choice
`p ?`	optional match
`p *`	zero or more repetitions
`p +`	one or more repetitions
`p^num`	exactly `num` repetitions
`p^+num`	at least `num` repetitions
`p^-num`	at most `num` repetitions
`(name <- p)`$^+$	grammar
`'string'`	literal string
`"string"`	literal string
`[class]`	character class
`.`	any character
`%name`	pattern `defs[name]` or a pre-defined pattern
`name`	non terminal
`<name>`	non terminal
`{}`	position capture
`{ p }`	simple capture
`{: p :}`	anonymous group capture
`{:name: p :}`	named group capture

`{~ p ~}`	substitution capture
`{| p |}`	table capture
`=name`	back reference
`p -> 'string'`	string capture
`p -> "string"`	string capture
`p -> num`	numbered capture
`p -> name`	function/query/string capture equivalent to `p / defs[name]`
`p => name`	match-time capture equivalent to `lpeg.Cmt(p, defs[name])`
`p ~> name`	fold capture (deprecated)
`p >> name`	accumulator capture equivalent to `(p % defs[name])`

Any space appearing in a syntax description can be replaced by zero or more space characters and Lua-style short comments (`--` until end of line).

Character classes define sets of characters. An initial `^` complements the resulting set. A range *x*`-`*y* includes in the set all characters with codes between the codes of *x* and *y*. A pre-defined class `%`*name* includes all characters of that class. A simple character includes itself in the set. The only special characters inside a class are `^` (special only if it is the first character); `]` (can be included in the set as the first character, after the optional `^`); `%` (special only if followed by a letter); and `-` (can be included in the set as the first or the last character).

Currently the pre-defined classes are similar to those from the Lua's string library (`%a` for letters, `%A` for non letters, etc.). There is also a class `%nl` containing only the newline character, which is particularly handy for grammars written inside long strings, as long strings do not interpret escape sequences like `\n`.

Functions

re.compile (string, [, defs])

Compiles the given string and returns an equivalent LPeg pattern. The given string may define either an expression or a grammar. The optional `defs` table provides extra Lua values to be used by the pattern.

re.find (subject, pattern [, init])

Searches the given pattern in the given subject. If it finds a match, returns the index where this occurrence starts and the index where it ends. Otherwise, returns nil.

An optional numeric argument `init` makes the search starts at that position in the subject string. As usual in Lua libraries, a negative value counts from the end.

re.gsub (subject, pattern, replacement)

Does a *global substitution*, replacing all occurrences of `pattern` in the given `subject` by `replacement`.

re.match (subject, pattern)

Matches the given pattern against the given subject, returning all captures.

re.updatelocale ()

Updates the pre-defined character classes to the current locale.

Some Examples

A complete simple program

The next code shows a simple complete Lua program using the `re` module:

```
local re = require"re"

-- find the position of the first numeral in a string
print(re.find("the number 423 is odd", "[0-9]+"))  --> 12    14

-- returns all words in a string
print(re.match("the number 423 is odd", "({%a+} / .)*"))
--> the    number    is    odd

-- returns the first numeral in a string
print(re.match("the number 423 is odd", "s <- {%d+} / . s"))
--> 423

-- substitutes a dot for each vowel in a string
print(re.gsub("hello World", "[aeiou]", "."))
--> h.ll. W.rld
```

Balanced parentheses

The following call will produce the same pattern produced by the Lua expression in the **balanced parentheses** example:

```
b = re.compile[[  balanced <- "(" ([^()] / balanced)* ")"  ]]
```

String reversal

The next example reverses a string:

```
rev = re.compile[[ R <- (!.) -> '' / ({.} R) -> '%2%1']]
print(rev:match"0123456789")   --> 9876543210
```

CSV decoder

The next example replicates the **CSV decoder**:

```
record = re.compile[[
  record <- {| field (',' field)* |} (%nl / !.)
  field <- escaped / nonescaped
  nonescaped <- { [^,"%nl]* }
  escaped <- '"' {~ ([^"] / '""' -> '"')* ~} '"'
]]
```

Lua's long strings

The next example matches Lua long strings:

```
c = re.compile([[
  longstring <- ('[' {:eq: '='* :} '[' close)
  close <- ']' =eq ']' / . close
]])

print(c:match'[==[]]===]]]]==]===[]')   --> 17
```

Abstract Syntax Trees

This example shows a simple way to build an abstract syntax tree (AST) for a given grammar. To keep our example simple, let us consider the following grammar for lists of names:

```
p = re.compile[[
      listname <- (name s)*
      name <- [a-z][a-z]*
      s <- %s*
]]
```

Now, we will add captures to build a corresponding AST. As a first step, the pattern will build a table to represent each non terminal; terminals will be represented by their corresponding strings:

```
c = re.compile[[
      listname <- {| (name s)* |}
      name <- {| {[a-z][a-z]*} |}
      s <- %s*
]]
```

Now, a match against `"hi hello bye"` results in the table `{{"hi"}, {"hello"}, {"bye"}}`.

For such a simple grammar, this AST is more than enough; actually, the tables around each single name are already overkilling. More complex grammars, however, may need some more structure. Specifically, it would be useful if each table had a `tag` field telling what non terminal that table represents. We can add such a tag using **named group captures**:

```
x = re.compile[[
      listname <- {| {:tag: '' -> 'list':} (name s)* |}
      name <- {| {:tag: '' -> 'id':} {[a-z][a-z]*} |}
      s <- ' '*
]]
```

With these group captures, a match against `"hi hello bye"` results in the following table:

```
{tag="list",
  {tag="id", "hi"},
  {tag="id", "hello"},
  {tag="id", "bye"}
}
```

Indented blocks

This example breaks indented blocks into tables, respecting the indentation:

```
p = re.compile[[
  block <- {| {:ident:' '*:} line
            ((=ident !' ' line) / &(=ident ' ') block)* |}
  line <- {[^%nl]*} %nl
]]
```

As an example, consider the following text:

```
t = p:match[[
first line
  subline 1
  subline 2
second line
third line
  subline 3.1
    subline 3.1.1
  subline 3.2
]]
```

The resulting table `t` will be like this:

```
   {'first line'; {'subline 1'; 'subline 2'; ident = '  '};
    'second line';
    'third line'; { 'subline 3.1'; {'subline 3.1.1'; ident = '    '};
                    'subline 3.2'; ident = '  '};
    ident = ''}
```

Macro expander

This example implements a simple macro expander. Macros must be defined as part of the pattern, following some simple rules:

```
p = re.compile[[
      text <- {~ item* ~}
      item <- macro / [^()] / '(' item* ')'
      arg <- ' '* {~ (!',' item)* ~}
      args <- '(' arg (',' arg)* ')'
      -- now we define some macros
      macro <- ('apply' args) -> '%1(%2)'
             / ('add' args) -> '%1 + %2'
             / ('mul' args) -> '%1 * %2'
]]

print(p:match"add(mul(a,b), apply(f,x))")   --> a * b + f(x)
```

A `text` is a sequence of items, wherein we apply a substitution capture to expand any macros. An `item` is either a macro, any character different from parentheses, or a parenthesized expression. A macro argument (`arg`) is a sequence of items different from a comma. (Note that a comma may appear inside an item, e.g., inside a parenthesized expression.) Again we do a substitution capture to expand any macro in the argument before expanding the outer macro. `args` is a list of arguments separated by commas. Finally we define the macros. Each macro is a string substitution; it replaces the macro name and its arguments by its corresponding string, with each `%`*n* replaced by the *n*-th argument.

Patterns

This example shows the complete syntax of patterns accepted by `re`.

```
p = [=[

pattern         <- exp !.
exp             <- S (grammar / alternative)

alternative     <- seq ('/' S seq)*
seq             <- prefix*
prefix          <- '&' S prefix / '!' S prefix / suffix
suffix          <- primary S (([+*?]
                           / '^' [+-]? num
                           / '->' S (string / '{}' / name)
                           / '>>' S name
                           / '=>' S name) S)*

primary         <- '(' exp ')' / string / class / defined
                 / '{:' (name ':')? exp ':}'
                 / '=' name
                 / '{}'
                 / '{~' exp '~}'
                 / '{|' exp '|}'
                 / '{' exp '}'
                 / '.'
                 / name S !arrow
                 / '<' name '>'          -- old-style non terminals

grammar         <- definition+
definition      <- name S arrow exp

class           <- '[' '^'? item (!']' item)* ']'
item            <- defined / range / .
range           <- . '-' [^]]

S               <- (%s / '--' [^%nl]*)*   -- spaces and comments
name            <- [A-Za-z_][A-Za-z0-9_]*
arrow           <- '<-'
num             <- [0-9]+
string          <- '"' [^"]* '"' / "'" [^']* "'"
defined         <- '%' name

]=]

print(re.match(p, p))   -- a self description must match itself
```

License

This module is part of the **LPeg** package and shares its **license**.