



# LPeg.re

## *Regex syntax for LPEG*

### Abstract

<+ Write Article Abstract +>

### Keywords

<+ Write Article Keywords +>  
Regex syntax for LPEG

### re

1. [Basic Constructions](#)
2. [Functions](#)
3. [Some Examples](#)
4. [License](#)

### 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
<code>( p )</code>	grouping
<code>'string'</code>	literal string
<code>"string"</code>	literal string
<code>[class]</code>	character class
<code>.</code>	any character
<code>%name</code>	pattern <code>defs[name]</code> or a pre-defined pattern
<code>&lt;name&gt;</code>	non terminal
<code>{ }</code>	position capture
<code>{ p }</code>	simple capture
<code>{: p :}</code>	anonymous group capture
<code>{:name: p :}</code>	named group capture
<code>{~ p ~}</code>	substitution capture
<code>=name</code>	back reference
<code>p ?</code>	optional match
<code>p *</code>	zero or more repetitions
<code>p +</code>	one or more repetitions
<code>p ^num</code>	exactly <code>n</code> repetitions

<code>p<sup>+</sup>num</code>	at least n repetitions
<code>p<sup>-</sup>num</code>	at most n repetitions
<code>p -&gt; 'string'</code>	string capture
<code>p -&gt; "string"</code>	string capture
<code>p -&gt; {}</code>	table capture
<code>p -&gt; name</code>	function/query/string capture equivalent to <code>p / defs[name]</code>
<code>p =&gt; name</code>	match-time capture equivalent to <code>lpeg.Cmt(p, defs[name])</code>
<code>&amp; p</code>	and predicate
<code>! p</code>	not predicate
<code>p1 p2</code>	concatenation
<code>p1 / p2</code>	ordered choice
<code>(name &lt;- p)<sup>+</sup></code>	grammar

Any space appearing in a syntax description can be replaced by zero or more space characters and Lua-style 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, plus the captures made by the pattern (if any). Otherwise, returns `nil`.

`re.match (subject, pattern)`. Matches the given pattern against the given subject.

`re.updatelocale ()`. Updates the pre-defined character classes to the current locale.

### Some Examples

**Balanced parentheses.** As a simple example, 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>) => void
  close <- ']' =eq '[' / . <close>
]], {void = function () return true end})

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

**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
  subtitle 1
  subtitle 2
second line
third line
  subtitle 3.1
    subtitle 3.1.1
  subtitle 3.2
]]
```

The resulting table `t` will be like this:

```
{'first line'; {'subtitle 1'; 'subtitle 2'; ident = ' '};
'second line';
'third line'; { 'subtitle 3.1'; {'subtitle 3.1.1'; ident = ' ' };
  'subtitle 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.

#### License

Copyright ©2008 Lua.org, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

\$Id: re.html,v 1.11 2008/10/10 18:14:06 roberto Exp \$

<http://www.inf.puc-rio.br/roberto/lpeg/re.html>