Regex syntax for LPEG

1 re

1.1 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
'string'	literal string
"string"	literal string
[class]	character class
.	any character
%name	pattern defs[name] or a pre-defined pattern
<name>	non terminal
{}	position capture
{ p }	simple capture
{: p :}	anonymous group capture
{:name: p :}	named group capture
{~ p ~}	substitution capture
=name	back reference
p ?	optional match
p *	zero or more repetitions
p +	one or more repetitions

`p^num`	exactly `n` repetitions
`p^+num`	at least `n` repetitions
`p^-num`	at most `n` repetitions
`p -> 'string'`	string capture
`p -> "string"`	string capture
`p -> {}`	table 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`	and predicate
`! p`	not predicate
`p1 p2`	concatenation
`p1 / p2`	ordered choice
`(name <- p)`$^+$	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`.

1.2 Functions

1.2.1 `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.

1.2.2 `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.

1.2.3 `re.match (subject, pattern)`

Matches the given pattern against the given subject.

1.2.4 `re.updatelocale ()`

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

1.3 Some Examples

1.3.1 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>)* ")"  ]]
```

1.3.2 String reversal

The next example reverses a string:

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

1.3.3 CSV decoder

The next example replicates the `CSV decoder`:

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

1.3.4 Lua's long strings

The next example mathes Lua long strings:

```
c = re.compile([[
  longstring <- ('[' {:eq: '='* :} '[' <close>) => void
  close <- ']' =eq ']' / . <close>
]], {void = function () return true end})

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

1.3.5 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 = ''}
```

1.3.6 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.

1.4 License

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