Lpeg Recipes

Lua recipes for LPeg (LuaPeg), a new pattern-matching library for Lua.

See LpegTutorial for an introduction and an [online LPeg grammar tester] for experimentation.

Place examples of Lua code using LPeg for parsing to help further the understanding of how to use parsing expression grammars.

Number Patterns

Written by Caleb Place. A table of number patterns to use for matching.

```
local number = {}

local digit = R("09")

-- Matches: 10, -10, 0
number.integer =
        (S("+-") ^ -1) *
        (digit   ^  1)

-- Matches: .6, .899, .9999873
number.fractional =
        (P(".")   ) *
        (digit ^ 1)

-- Matches: 55.97, -90.8, .9
number.decimal =
        (number.integer *                     -- Integer
        (number.fractional ^ -1)) +           -- Fractional
        ((S("+-") ^ -1) * number.fractional)  -- Completely fractional number

-- Matches: 60.9e07, 9e-4, 681E09
number.scientific =
        number.decimal * -- Decimal number
        S("Ee") *        -- E or e
        number.integer   -- Exponent

-- Matches all of the above
number.number =
        number.scientific + number.decimal -- Decimal number allows for everything else, and scientific matches scientific
```

C Comment Parser

```
local BEGIN_COMMENT = lpeg.P("/*")
local END_COMMENT = lpeg.P("*/")
local NOT_BEGIN = (1 - BEGIN_COMMENT)^0
local NOT_END = (1 - END_COMMENT)^0
local FULL_COMMENT_CONTENTS = BEGIN_COMMENT * NOT_END * END_COMMENT

-- Parser to find comments from a string
local searchParser = (NOT_BEGIN * lpeg.C(FULL_COMMENT_CONTENTS))^0
-- Parser to find non-comments from a string
local filterParser = (lpeg.C(NOT_BEGIN) * FULL_COMMENT_CONTENTS)^0 * lpeg.C(NOT_BEGIN)

-- Simpler version, although empirically it is slower.... (why?) ... any optimization
-- suggestions are desired as well as optimum integration w/ C++ comments and other
-- syntax elements
local searchParser = (lpeg.C(FULL_COMMENT_CONTENTS) + 1)^0
-- Suggestion by Roberto to make the search faster
-- Works because it loops fast over all non-slashes, then it begins the slower match phase
local searchParser = ((1 - lpeg.P"/")^0 * (lpeg.C(FULL_COMMENT_CONTENTS) + 1))^0
```

Evaluate Standard Roman Numerals

The numeral is given in the variable `text`.

```
do
local add = function (x,y) return x+y end
local P,Ca,Cc= lpeg.P,lpeg.Ca,lpeg.Cc
local symbols = { I=1,V=5,X=10,L=50,C=100,D=500,M=1000,
```

```
   IV=4,IX=9,XL=40,XC=90,CD=400,CM=900}
local env = getfenv(1)
for s,n in pairs(symbols) do env[s:lower()] = P(s)*Cc(n)/add end
setfenv(1,env)
local MS = m^0
local CS = (d*c^(-4)+cd+cm+c^(-4))^(-1)
local XS = (l*x^(-4)+xl+xc+x^(-4))^(-1)
local IS = (v*i^(-4)+ix+iv+i^(-4))^(-1)
local p = Ca(Cc(0)*MS*CS*XS*IS)
local result = p:match(text:upper())
print(result or "?")
end
```

Match Sequences of Consecutive Integers

Needs Lpeg version 8.

```
do
local C,Cb,Cmt,R,S = lpeg.C,lpeg.Cb,lpeg.Cmt,lpeg.R,lpeg.S
local some = function (p) return (p+1)^1 end
local digit,space = R "09",S " "
local num = C(digit^1)/tonumber
local check = Cmt(Cb(1)*num,function (s,i,x,y)
      if y == x+1 then return i,y end end)
local monotone = some(C(num*(space^1*check)^0))
local m = monotone:match(text)
print (m or "?")
end
```

Match a list of integers or ranges

Recognise a list of integer values or ranges of integer values.

Return a table containing

- for each individual integer, a table of length 1
- for each range, a table of length 2, giving the limits of the range

Return nil if no integer values or ranges are found

Examples

- '1' --> { {1} }
- '1-5' --> { {1, 5} }
- '1,2,3' --> { {1}, {2}, {3} }
- '1,4-8' --> { {1}, {4, 8 } }
- '4-8,4,8' --> { {4, 8}, {4}, {8} }
- '6-2' --> { {6, 2} }

```
local re = require 're'

local list_parser = re.compile [[
   list <- ( singleint_or_range ( ',' singleint_or_range ) * ) -> {}
   singleint_or_range <- range / singleint
   singleint <- { int } -> {}
   range <- ( { int } '-' { int } ) -> {}
   int <- %d+
]]

local function parse_list(list_string)
   local t = list_parser:match(list_string)
   -- further processing to remove overlaps, duplicates, sort into ascending order, etc
   return t
end
```

Match a fixed number of repetitions of a pattern

Matching a precise number of occurrences of a certain pattern. -- ValeriuPalos?

```
function multiply_pattern(item, count)
    return lpeg.Cmt(lpeg.P(true),
                function(s, i)
                    local set, offset = {}, i
                    for j = 1, count do
                        set[j], offset = lpeg.match(item * lpeg.Cp(), s, offset)
                        if not offset then
                            return false
```

```
                    end
                end
                return offset, set
            end)
end
```

A detailed explanation [is described here] along with a method to match between a minimum and a maximum number of pattern occurrences.

Lua Lexer

This is a Lua lexer in LPeg. The original author is PeterOdding. This lexer eventually became [LXSH] which includes Lua and C lexers and syntax highlighters.

```
--[[

= ABOUT
This module uses Roberto Ierusalimschy's powerful new pattern matching library
LPeg[1] to tokenize Lua source-code in to a table of tokens. I think it handles
all of Lua's syntax, but if you find anything missing I would appreciate a mail
at peter@peterodding.com. This lexer is based on the BNF[2] from the Lua manual.

= USAGE
I've saved my copy of this module under [$LUA_PATH/lexers/lua.lua] which means
I can use it like in the following interactive prompt:

   Lua 5.1.1  Copyright (C) 1994-2006 Lua.org, PUC-Rio
   > require 'lexers.lua'
   > tokens = lexers.lua [=[
   >> 42 or 0
   >> -- some Lua source-code in a string]=]
   > = tokens
   table: 00422E40
   > lexers.lua.print(tokens)
   line 1, number: `42`
   line 1, whitespace: ` `
   line 1, keyword: `or`
   line 1, whitespace: ` `
   line 1, number: `0`
   line 1, whitespace: `
   `
   line 2, comment: `-- some Lua source-code in a string`
   total of 7 tokens, 2 lines

The returned table [tokens] looks like this:

{
   -- type       , text, line
   { 'number'    , '42', 1 },
   { 'whitespace', ' ' , 1 },
   { 'keyword'   , 'or', 1 },
   { 'whitespace', ' ' , 1 },
   { 'number'    , '0' , 1 },
   { 'whitespace', '\n', 1 },
   { 'comment'   , '-- some Lua source-code in a string', 2 },
}

= CREDITS
Written by Peter Odding, 2007/04/04

= THANKS TO
- the Lua authors for a wonderful language;
- Roberto for LPeg;
- caffeine for keeping me awake :)

= LICENSE
Shamelessly ripped from the SQLite[3] project:

   The author disclaims copyright to this source code.  In place of a legal
   notice, here is a blessing:

      May you do good and not evil.
      May you find forgiveness for yourself and forgive others.
      May you share freely, never taking more than you give.

[1] http://www.inf.puc-rio.br/~roberto/lpeg.html
[2] http://lua.org/manual/5.1/manual.html#8
[3] http://sqlite.org

--]]

-- since this module is intended to be loaded with require() we receive the
-- name used to load us in ... and pass it on to module()
module(..., package.seeall)

-- written for LPeg .5, by the way
local lpeg = require 'lpeg'
local P, R, S, C, Cc, Ct = lpeg.P, lpeg.R, lpeg.S, lpeg.C, lpeg.Cc, lpeg.Ct
```

```
-- create a pattern which captures the lua value [id] and the input matching
-- [patt] in a table
local function token(id, patt) return Ct(Cc(id) * C(patt)) end

local digit = R('09')

-- range of valid characters after first character of identifier
local idsafe = R('AZ', 'az', '\127\255') + P '_'

-- operators
local operator = token('operator', P '==' + P '~=' + P '<=' + P '>=' + P '...'
                                   + P '..' + S '+-*/%^#=<>;:,.{}[]()')
-- identifiers
local ident = token('identifier', idsafe * (idsafe + digit + P '.') ^ 0)

-- keywords
local keyword = token('keyword', (P 'and' + P 'break' + P 'do' + P 'else' +
   P 'elseif' + P 'end' + P 'false' + P 'for' + P 'function' + P 'if' +
   P 'in' + P 'local' + P 'nil' + P 'not' + P 'or' + P 'repeat' + P 'return' +
   P 'then' + P 'true' + P 'until' + P 'while') * -(idsafe + digit))

-- numbers
local number_sign = S'+-'^-1
local number_decimal = digit ^ 1
local number_hexadecimal = P '0' * S 'xX' * R('09', 'AF', 'af') ^ 1
local number_float = (digit^1 * P'.' * digit^0 + P'.' * digit^1) *
                     (S'eE' * number_sign * digit^1)^-1
local number = token('number', number_hexadecimal +
                               number_float +
                               number_decimal)

-- callback for [=[ long strings ]=]
-- ps. LPeg is for Lua what regex is for Perl, which makes me smile :)
local longstring = #(P '[[' + (P '[' * P '=' ^ 0 * P '['))
local longstring = longstring * P(function(input, index)
   local level = input:match('^%[(=*)%[', index)
   if level then
      local _, stop = input:find(']' .. level .. ']', index, true)
      if stop then return stop + 1 end
   end
end)

-- strings
local singlequoted_string = P "'" * ((1 - S "'\r\n\f\\") + (P '\\' * 1)) ^ 0 * "'"
local doublequoted_string = P '"' * ((1 - S '"\r\n\f\\') + (P '\\' * 1)) ^ 0 * '"'
local string = token('string', singlequoted_string +
                               doublequoted_string +
                               longstring)

-- comments
local singleline_comment = P '--' * (1 - S '\r\n\f') ^ 0
local multiline_comment = P '--' * longstring
local comment = token('comment', multiline_comment + singleline_comment)

-- whitespace
local whitespace = token('whitespace', S('\r\n\f\t ')^1)

-- ordered choice of all tokens and last-resort error which consumes one character
local any_token = whitespace + number + keyword + ident +
                  string + comment + operator + token('error', 1)

-- private interface
local table_of_tokens = Ct(any_token ^ 0)

-- increment [line] by the number of line-ends in [text]
local function sync(line, text)
   local index, limit = 1, #text
   while index <= limit do
      local start, stop = text:find('\r\n', index, true)
      if not start then
         start, stop = text:find('[\r\n\f]', index)
         if not start then break end
      end
      index = stop + 1
      line = line + 1
   end
   return line
end

-- we only need to synchronize the line-counter for these token types
local multiline_tokens = { comment = true, string = true, whitespace = true }

-- public interface
getmetatable(getfenv(1)).__call = function(self, input)
   assert(type(input) == 'string', 'bad argument #1 (expected string)')
   local line = 1
   local tokens = lpeg.match(table_of_tokens, input)
   for i, token in pairs(tokens) do
      token[3] = line
      if multiline_tokens[token[1]] then line = sync(line, token[2]) end
   end
```

```
   return tokens
end

-- if you really want to try it out before writing any code :P
function print(tokens)
   local print, format = _G.print, _G.string.format
   for _, token in pairs(tokens) do
      print(format('line %i, %s: `%s`', token[3], token[1], token[2]))
   end
   print(format('total of %i tokens, %i lines', #tokens, tokens[#tokens][3]))
end
```

Lua Parser

A Lua 5.1 parser in LPeg. Improvements welcome. -- Patrick Donnelly (batrick)

```
local lpeg = require "lpeg";

local locale = lpeg.locale();

local P, S, V = lpeg.P, lpeg.S, lpeg.V;

local C, Cb, Cc, Cg, Cs, Cmt =
    lpeg.C, lpeg.Cb, lpeg.Cc, lpeg.Cg, lpeg.Cs, lpeg.Cmt;

local shebang = P "#" * (P(1) - P "\n")^0 * P "\n";

local function K (k) -- keyword
  return P(k) * -(locale.alnum + P "_");
end

local lua = P {
  (shebang)^-1 * V "space" * V "chunk" * V "space" * -P(1);

  -- keywords

  keywords = K "and" + K "break" + K "do" + K "else" + K "elseif" +
             K "end" + K "false" + K "for" + K "function" + K "if" +
             K "in" + K "local" + K "nil" + K "not" + K "or" + K "repeat" +
             K "return" + K "then" + K "true" + K "until" + K "while";

  -- longstrings

  longstring = P { -- from Roberto Ierusalimschy's lpeg examples
    V "open" * C((P(1) - V "closeeq")^0) *
        V "close" / function (o, s) return s end;

    open = "[" * Cg((P "=")^0, "init") * P "[" * (P "\n")^-1;
    close = "]" * C((P "=")^0) * "]";
    closeeq = Cmt(V "close" * Cb "init", function (s, i, a, b) return a == b end)
  };

  -- comments & whitespace

  comment = P "--" * V "longstring" +
            P "--" * (P(1) - P "\n")^0 * (P "\n" + -P(1));

  space = (locale.space + V "comment")^0;

  -- Types and Comments

  Name = (locale.alpha + P "_") * (locale.alnum + P "_")^0 - V "keywords";
  Number = (P "-")^-1 * V "space" * P "0x" * locale.xdigit^1 *
               -(locale.alnum + P "_") +
           (P "-")^-1 * V "space" * locale.digit^1 *
               (P "." * locale.digit^1)^-1 * (S "eE" * (P "-")^-1 *
                   locale.digit^1)^-1 * -(locale.alnum + P "_") +
           (P "-")^-1 * V "space" * P "." * locale.digit^1 *
               (S "eE" * (P "-")^-1 * locale.digit^1)^-1 *
               -(locale.alnum + P "_");
  String = P "\"" * (P "\\" * P(1) + (1 - P "\""))^0 * P "\"" +
           P "'" * (P "\\" * P(1) + (1 - P "'"))^0 * P "'" +
           V "longstring";

  -- Lua Complete Syntax

  chunk = (V "space" * V "stat" * (V "space" * P ";")^-1)^0 *
              (V "space" * V "laststat" * (V "space" * P ";")^-1)^-1;

  block = V "chunk";

  stat = K "do" * V "space" * V "block" * V "space" * K "end" +
         K "while" * V "space" * V "exp" * V "space" * K "do" * V "space" *
             V "block" * V "space" * K "end" +
         K "repeat" * V "space" * V "block" * V "space" * K "until" *
             V "space" * V "exp" +
         K "if" * V "space" * V "exp" * V "space" * K "then" *
             V "space" * V "block" * V "space" *
             (K "elseif" * V "space" * V "exp" * V "space" * K "then" *
```

```
            V "space" * V "block" * V "space"
           )^0 *
           (K "else" * V "space" * V "block" * V "space")^-1 * K "end" +
       K "for" * V "space" * V "Name" * V "space" * P "=" * V "space" *
           V "exp" * V "space" * P "," * V "space" * V "exp" *
           (V "space" * P "," * V "space" * V "exp")^-1 * V "space" *
           K "do" * V "space" * V "block" * V "space" * K "end" +
       K "for" * V "space" * V "namelist" * V "space" * K "in" * V "space" *
           V "explist" * V "space" * K "do" * V "space" * V "block" *
           V "space" * K "end" +
       K "function" * V "space" * V "funcname" * V "space" *  V "funcbody" +
       K "local" * V "space" * K "function" * V "space" * V "Name" *
           V "space" * V "funcbody" +
       K "local" * V "space" * V "namelist" *
           (V "space" * P "=" * V "space" * V "explist")^-1 +
       V "varlist" * V "space" * P "=" * V "space" * V "explist" +
       V "functioncall";

laststat = K "return" * (V "space" * V "explist")^-1 + K "break";

funcname = V "Name" * (V "space" * P "." * V "space" * V "Name")^0 *
    (V "space" * P ":" * V "space" * V "Name")^-1;

namelist = V "Name" * (V "space" * P "," * V "space" * V "Name")^0;

varlist = V "var" * (V "space" * P "," * V "space" * V "var")^0;

-- Let's come up with a syntax that does not use left recursion
-- (only listing changes to Lua 5.1 extended BNF syntax)
-- value ::= nil | false | true | Number | String | '...' | function |
--           tableconstructor | functioncall | var | '(' exp ')'
-- exp ::= unop exp | value [binop exp]
-- prefix ::= '(' exp ')' | Name
-- index ::= '[' exp ']' | '.' Name
-- call ::= args | ':' Name args
-- suffix ::= call | index
-- var ::= prefix {suffix} index | Name
-- functioncall ::= prefix {suffix} call

-- Something that represents a value (or many values)
value = K "nil" +
        K "false" +
        K "true" +
        V "Number" +
        V "String" +
        P "..." +
        V "function" +
        V "tableconstructor" +
        V "functioncall" +
        V "var" +
        P "(" * V "space" * V "exp" * V "space" * P ")";

-- An expression operates on values to produce a new value or is a value
exp = V "unop" * V "space" * V "exp" +
      V "value" * (V "space" * V "binop" * V "space" * V "exp")^-1;

-- Index and Call
index = P "[" * V "space" * V "exp" * V "space" * P "]" +
        P "." * V "space" * V "Name";
call = V "args" +
       P ":" * V "space" * V "Name" * V "space" * V "args";

-- A Prefix is a the leftmost side of a var(iable) or functioncall
prefix = P "(" * V "space" * V "exp" * V "space" * P ")" +
         V "Name";
-- A Suffix is a Call or Index
suffix = V "call" +
         V "index";

var = V "prefix" * (V "space" * V "suffix" * #(V "space" * V "suffix"))^0 *
          V "space" * V "index" +
      V "Name";
functioncall = V "prefix" *
                   (V "space" * V "suffix" * #(V "space" * V "suffix"))^0 *
               V "space" * V "call";

explist = V "exp" * (V "space" * P "," * V "space" * V "exp")^0;

args = P "(" * V "space" * (V "explist" * V "space")^-1 * P ")" +
       V "tableconstructor" +
       V "String";

["function"] = K "function" * V "space" * V "funcbody";

funcbody = P "(" * V "space" * (V "parlist" * V "space")^-1 * P ")" *
               V "space" *  V "block" * V "space" * K "end";

parlist = V "namelist" * (V "space" * P "," * V "space" * P "...")^-1 +
          P "...";

tableconstructor = P "{" * V "space" * (V "fieldlist" * V "space")^-1 * P "}";
```

```
  fieldlist = V "field" * (V "space" * V "fieldsep" * V "space" * V "field")^0
                * (V "space" * V "fieldsep")^-1;

  field = P "[" * V "space" * V "exp" * V "space" * P "]" * V "space" * P "=" *
           V "space" * V "exp" +
          V "Name" * V "space" * P "=" * V "space" * V "exp" +
          V "exp";

  fieldsep = P "," +
             P ";";

  binop = K "and" + -- match longest token sequences first
          K "or" +
          P ".." +
          P "<=" +
          P ">=" +
          P "==" +
          P "~=" +
          P "+" +
          P "-" +
          P "*" +
          P "/" +
          P "^" +
          P "%" +
          P "<" +
          P ">";

  unop = P "-" +
         P "#" +
         K "not";
};
```

Also see LuaFish, Leg[1], or the [Lua parser in trolledit].

C Lexer

This lexes ANSI C. Improvements welcome. --DavidManura

```
-- Lua LPeg lexer for C.
-- Note:
--   Does not handle C preprocessing macros.
--   Not well tested.
--
-- David Manura, 2007, public domain.  Based on ANSI C Lex
--   specification in http://www.quut.com/c/ANSI-C-grammar-l-1998.html
--   (Jutta Degener, 2006; Tom Stockfisch, 1987, Jeff Lee, 1985)

local lpeg = require 'lpeg'

local P, R, S, C =
  lpeg.P, lpeg.R, lpeg.S, lpeg.C

local whitespace = S' \t\v\n\f'

local digit = R'09'
local letter = R('az', 'AZ') + P'_'
local alphanum = letter + digit
local hex = R('af', 'AF', '09')
local exp = S'eE' * S'+-'^-1 * digit^1
local fs = S'fFlL'
local is = S'uUlL'^0

local hexnum = P'0' * S'xX' * hex^1 * is^-1
local octnum = P'0' * digit^1 * is^-1
local decnum = digit^1 * is^-1
local floatnum = digit^1 * exp * fs^-1 +
                 digit^0 * P'.' * digit^1 * exp^-1 * fs^-1 +
                 digit^1 * P'.' * digit^0 * exp^-1 * fs^-1
local numlit = hexnum + octnum + floatnum + decnum

local charlit =
  P'L'^-1 * P"'" * (P'\\' * P(1) + (1 - S"\\'"))^1 * P"'"

local stringlit =
  P'L'^-1 * P'"' * (P'\\' * P(1) + (1 - S'\\"'))^0 * P'"'

local ccomment = P'/*' * (1 - P'*/')^0 * P'*/'
local newcomment = P'//' * (1 - P'\n')^0
local comment = (ccomment + newcomment)
              / function(...) print('COMMENT', ...) end

local literal = (numlit + charlit + stringlit)
              / function(...) print('LITERAL', ...) end

local keyword = C(
  P"auto" +
  P"_Bool" +
  P"break" +
```

```
  P"case" +
  P"char" +
  P"_Complex" +
  P"const" +
  P"continue" +
  P"default" +
  P"do" +
  P"double" +
  P"else" +
  P"enum" +
  P"extern" +
  P"float" +
  P"for" +
  P"goto" +
  P"if" +
  P"_Imaginary" +
  P"inline" +
  P"int" +
  P"long" +
  P"register" +
  P"restrict" +
  P"return" +
  P"short" +
  P"signed" +
  P"sizeof" +
  P"static" +
  P"struct" +
  P"switch" +
  P"typedef" +
  P"union" +
  P"unsigned" +
  P"void" +
  P"volatile" +
  P"while"
) / function(...) print('KEYWORD', ...) end

local identifier = (letter * alphanum^0 - keyword * (-alphanum))
                 / function(...) print('ID',...) end

local op = C(
  P"..." +
  P">>=" +
  P"<<=" +
  P"+=" +
  P"-=" +
  P"*=" +
  P"/=" +
  P"%=" +
  P"&=" +
  P"^=" +
  P"|=" +
  P">>" +
  P"<<" +
  P"++" +
  P"--" +
  P"->" +
  P"&&" +
  P"||" +
  P"<=" +
  P">=" +
  P"==" +
  P"!=" +
  P";" +
  P"{" + P"<%" +
  P"}" + P"%>" +
  P"," +
  P":" +
  P"=" +
  P"(" +
  P")" +
  P"[" + P"<:" +
  P"]" + P":>" +
  P"." +
  P"&" +
  P"!" +
  P"~" +
  P"-" +
  P"+" +
  P"*" +
  P"/" +
  P"%" +
  P"<" +
  P">" +
  P"^" +
  P"|" +
  P"?"
) / function(...) print('OP', ...) end

local tokens = (comment + identifier + keyword +
                literal + op + whitespace)^0

-- frontend
```

```
local filename = arg[1]
local fh = assert(io.open(filename))
local input = fh:read'*a'
fh:close()
print(lpeg.match(tokens, input))
```

~~ ThomasHarningJr : Suggestion for optimization of the 'op' matcher in the C preprocessor... This should be faster due to the use of sets instead of making tons of 'basic' string comparisons. Not sure 'how' much faster...

```
local shiftOps = P">>" + P"<<"
local digraphs = P"<%" + P"%>" + P"<:" + P":>" -- {, }, [, ]
local op = C(
-- First match the multi-char items
  P"..." +
  ((shiftOps + S("+-*/%&^|<>=!")) * P"=") +
  shiftOps +
  P"++" +
  P"--" +
  P"&&" +
  P"||" +
  P"->" +
  digraphs +
  S(";{},:=()[].&!~-+*/%<>^|?")
) / function(...) print('OP', ...) end
```

See also Peter "Corsix" Cawley's https://github.com/CorsixTH/CorsixTH/blob/master/LDocGen/c_tokenise.lua and the [C parser in trolledit].

C Parser

[ceg] - Wesley Smith's C99 parser

XML Parser

See the [XML parser in trolledit].

SciTE Lexers

[Scintillua] supports LPeg lexers. A number of [examples] are included.

Parsing UTF-8

Like Lua itself, LPeG only works with single bytes, not potentially-multibyte characters (which can occur in UTF-8). Here are some tricks that help you parse UTF-8 text.

lpeg.S()

The set function assumes that every byte is a character, so you can't use it to match UTF-8 characters. However, you can emulate it with the + operator.

```
local currency_symbol = lpeg.P('$') + lpeg.P('£') + lpeg.P('¥') + lpeg.P('¢')
```

lpeg.R()

Likewise, the range operator works on single bytes only, so it cannot be used to match UTF-8 characters outside ASCII.

Character classes

The character classes provided by lpeg.locale() only work on single bytes, even under a UTF-8 locale. By using [ICU4Lua], you can create equivalent character classes which will match UTF-8 characters (regardless of the current locale):

```
-- lpeg_unicode_locale.lua

local lpeg = require 'lpeg'
local U    = require 'icu.ustring'
local re   = require 'icu.regex'

local utf8_codepoint
do
  --[=[
  Valid UTF-8 sequences (excluding isolated surrogates):
```

```
  Code points        Bit patterns                          Hexadecimal
  -----------------  ------------------------------------  -----------------------------
    U+0000-U+007F    0xxxxxxx                              00-7F
    U+0080-U+07FF    110xxxxx 10xxxxxx                     C2-DF  80-BF
    U+0800-U+0FFF    11100000 101xxxxx 10xxxxxx            E0     A0-BF(*)  80-BF
    U+1000-U+1FFF    11100001 10xxxxxx 10xxxxxx            E1     80-BF     80-BF
    U+2000-U+3FFF    1110001x 10xxxxxx 10xxxxxx            E2-E3  80-BF     80-BF
    U+4000-U+7FFF    111001xx 10xxxxxx 10xxxxxx            E4-E7  80-BF     80-BF
    U+8000-U+BFFF    111010xx 10xxxxxx 10xxxxxx            E8-EB  80-BF     80-BF
    U+C000-U+CFFF    11101100 10xxxxxx 10xxxxxx            EC     80-BF     80-BF
    U+D000-U+D7FF    11101101 100xxxxx 10xxxxxx            ED     80-9F(*)  80-BF
    U+E000-U+FFFF    1110111x 10xxxxxx 10xxxxxx            EE-EF  80-BF     80-BF
   U+10000-U+1FFFF   11110000 1001xxxx 10xxxxxx 10xxxxxx  F0     90-BF(*)  80-BF  80-BF
   U+20000-U+3FFFF   11110000 101xxxxx 10xxxxxx 10xxxxxx  F0     80-BF     80-BF  80-BF
   U+40000-U+7FFFF   11110001 10xxxxxx 10xxxxxx 10xxxxxx  F1     80-BF     80-BF  80-BF
   U+80000-U+FFFFF   1111001x 10xxxxxx 10xxxxxx 10xxxxxx  F2-F3  80-BF     80-BF  80-BF
  U+100000-U+10FFFF  11110100 1000xxxx 10xxxxxx 10xxxxxx  F4     80-8F(*)  80-BF  80-BF
  -----------------  ------------------------------------  -----------------------------
  Leading byte values in 0xC0-0xC1 and 0xF5-0xFF are always invalid.
  The first continuation byte (for UTF-8 sequences of 3 or 4 bytes) has a more
  restricted range of validity (*), depending on the leading byte.
  --]=]

  -- decode a two-byte UTF-8 sequence
  local function f2 (s)
    local c1, c2 = string.byte(s, 1, 2)
    return c1 * 64 + c2 - 12416
  end

  -- decode a three-byte UTF-8 sequence
  local function f3 (s)
    local c1, c2, c3 = string.byte(s, 1, 3)
    return (c1 * 64 + c2) * 64 + c3 - 925824
  end

  -- decode a four-byte UTF-8 sequence
  local function f4 (s)
    local c1, c2, c3, c4 = string.byte(s, 1, 4)
    return ((c1 * 64 + c2) * 64 + c3) * 64 + c4 - 63447168
  end

  local cont = lpeg.R("\128\191") -- unrestricted continuation byte
  utf8_codepoint =
       lpeg.R("\000\127") / string.byte
    +  lpeg.R("\194\223") * cont / f2
    + (lpeg.P("\224")     * lpeg.R("\160\191") +
       lpeg.R("\225\238") * cont +
       lpeg.P("\239")     * lpeg.R("\128\159")) * cont / f3
    + (lpeg.P("\240")     * lpeg.R("\144\191") +
       lpeg.R("\241\243") * cont +
       lpeg.P("\244")     * lpeg.R("\128\143")) * cont * cont / f4
end

local cntrl  = re.compile('^\\p{cntrl}$')
local print  = re.compile('^\\p{print}$')
local space  = re.compile('^\\p{space}$')
local graph  = re.compile('^\\p{graph}$')
local punct  = re.compile('^\\p{punct}$')
local alnum  = re.compile('^\\p{alnum}$')
local digit  = re.compile('^\\p{digit}$')
local alpha  = re.compile('^\\p{alpha}$')
local upper  = re.compile('^\\p{upper}$')
local lower  = re.compile('^\\p{lower}$')
local xdigit = re.compile('^\\p{xdigit}$')

return {
  cntrl  = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(cntrl,  U.char(c)) end);
  print  = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(print,  U.char(c)) end);
  space  = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(space,  U.char(c)) end);
  graph  = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(graph,  U.char(c)) end);
  punct  = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(punct,  U.char(c)) end);
  alnum  = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(alnum,  U.char(c)) end);
  digit  = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(digit,  U.char(c)) end);
  alpha  = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(alpha,  U.char(c)) end);
  upper  = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(upper,  U.char(c)) end);
  lower  = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(lower,  U.char(c)) end);
  xdigit = lpeg.Cmt(utf8_codepoint, function(s,i,c) return not not re.match(xdigit, U.char(c)) end);
}
```

In your code, you might use it like this:

```
local lpeg = require 'lpeg'
local utf8 = require 'lpeg_utf8_locale'

local EOF      = lpeg.P(-1)
local word     = lpeg.C(utf8.alnum^1)
local tokenise = (word * (utf8.space^1 + EOF))^0 * EOF

print(tokenise:match('þetta eru æðisleg orð'))
```

Date/Time

https://mozilla-services.github.io/lua_sandbox_extensions/lpeg/modules/lpeg/date_time.html

LPeg Grammar Tester: http://lpeg.trink.com/share/date_time

Common Log Format

https://mozilla-services.github.io/lua_sandbox_extensions/lpeg/modules/lpeg/common_log_format.html

Nginx meta grammar generator: http://lpeg.trink.com/share/clf

---

Rsyslog

https://mozilla-services.github.io/lua_sandbox_extensions/syslog/modules/lpeg/syslog.html

Rsyslog meta grammar generator: http://lpeg.trink.com/share/syslog

---

IP Address

https://mozilla-services.github.io/lua_sandbox_extensions/lpeg/modules/lpeg/ip_address.html

---