```lua
-- Week 2, Activity 1:
-- Mastering LPeg

-- LPeg is a pattern-matching library for Lua, based on Parsing Expression
-- Grammars (PEG).

-- Pattern matching is a system for finding and extracting pieces of
-- information from a text. Most pattern-matching systems are based on
-- regexes (most regex systems are extensions of the original
-- regular-expression definition that break the nice properties of the
-- original, so we'll use the term "regex" for those extensions systems
-- and save "regular expression" for the original).

-- A "regex" is a string that specifies a pattern and occasionally what
-- to extract from the match. The pattern in a regex specifies the
-- "capture", which is what to extract from the match.

-- LPeg is not based on regexes, is firmly ground on PEGs, treating
-- patterns as first-class objects. The result is usually much more
-- verbose, but allow us to create patterns programmatically.

-- 1. THE BASICS:
-----------------

-- 1.1. lpeg.P:
---------------
local lpeg = require "lpeg"
local pt = require "pt"

-- creates a pattern with lpeg.P function
local p = lpeg.P("hello")

-- we can use lpeg.match funtion to match the pattern against subject:
print(lpeg.match(p, "hello world"))                --> 6

-- or we can use the match metod from the pattern against subject:
print(p:match("hello world"))                      --> 6

print(lpeg.match(p, "hi world"))                   --> nil
print(p:match("hi world"))                         --> nil

-- lpeg.P matches a string literally ane return the position after the
-- last character that matched, oterwise, returns nil.
-- CAUTION: the "match" method don't search for the pattern, it tries to
-- match only at first position of the subject, doing what is called an
-- ANCHORED MATCH.

-- lpeg.P does not handle any magic character, all characters represent
-- themselves in a string.

-- When called with a positive integer, lpeg.P returns a pattern that matches
-- that number of characters, whatever they are:
local p = lpeg.P(3)
print(p:match("@ 4banana"))                        --> 4
print(p:match("ab"))                               --> nil (do not have 3 chars)

-- lpeg.P also accepts booleans:
--    lpeg.P(true), lpeg.P(""), lpeg.P(0): always succeeds, without consuming
--                                         any input
--    lpeg.P(false), lpeg.S(""), lpeg.R("za"):  always fails
```

```lua
-- 1.2. lpeg.S:
---------------
local p = lpeg.S("aeiou")

print(p:match("hello"))                                 --> nil (anchored match!)
print(p:match("all"))                                   --> 2

-- lpeg.S (S for set) receives a string and returns a pattern that matches a
-- SINGLE ocurrence os any character in that string, but as an anchored match.


-- 1.3. lpeg.R:
---------------
local p = lpeg.R("09")

print(p:match("hello"))                                 --> nil
print(p:match("42"))                                    --> 2 (only the first 4)
print(p:match("4z"))                                    --> 2 (only the first 4)
print(p:match("z4"))                                    --> nil (anchored match!)

-- lpeg.R (R for range) receives a two-charecter string, representing a range,
-- and returns a pattern that matches a single ocurrence of any character in
-- that range. lpeg.R can be called with several intervals, each represented
-- by a two-character string.

local p = lpeg.R("az", "AZ", "09")

print(p:match("banana"))                                --> 2
print(p:match("Zebra"))                                 --> 2
print(p:match("42 is the answer"))                      --> 2
print(p:match("@uvv"))                                  --> nil
print(p:match(" banana"))                               --> nil


-- 1.4. lpeg.locale:
--------------------
loc = lpeg.locale()

print(loc.space:match(" "))                             --> 2

-- lpeg.locale creates a set of patterns according to the current locale, and
-- returns them all in a table. The following patterns are created by
-- lpeg.locale:
--      spaces
--      alnum
--      alpha
--      cntrl
--      digit
--      graph
--      lower
--      print
--      punct
--      upper
--      xdigit
```

```lua
-- 1.5. Combining patterns:
---------------------------
-- It is possible to combine multiples patterns in several ways. As a first
-- example, the star operator * represents concatenation of patterns:
local p = lpeg.P(3) * lpeg.P("hi")
print(p:match("my hi"))                          --> 6
print(p:match("his hi"))                         --> nil


-- 1.6. Unicode:
----------------
-- Most parts of LPeg work for Unicode, including:
--   literals
--   concatenations
--   repetition
--   predicates

-- But some parts are restricted to ASCII:
--   sets
--   ranges
--   lpeg.P(1) matches 1 byte, not one unicode character



-- 2. REPETITIONS AND CHOICES:
------------------------------

-- 2.1. Pattern raised to a positive integer N:
-----------------------------------------------
-- A pattern raised to a positive integer N results in a new pattern that
-- behaves like the original pattern repeated N or more times.
loc = lpeg.locale()

p = loc.space^0 * lpeg.P("oi")                   -- 0+ spaces followed by "oi"
print(p:match("oi"))                             --> 3
print(p:match(" oi"))                            --> 4
print(p:match("  oi"))                           --> 5

-- CAUTION: repetitions in LPeg are always POSSESSIVE, they match as many
-- characters as possible, regardless of what comes next. The following pattern
-- always fails, for example, because the initial repetition will match the
-- whole string, leaving nothing to match the trailing "a":
p = lpeg.P(1)^0 * lpeg.P("a")
print(p:match("zzzzzzza"))                       --> always FAILS with nil

-- It is possible to EXCLUDE some character from the set created by lpeg.P(1):
noSC = lpeg.P(1) - ";"
p = noSC^0 * lpeg.P(";") * noSC^0 * lpeg.P(";")
print(p:match("one;two"))                        --> nil
print(p:match("one;two;"))                       --> 9
print(p:match(";;"))                             --> 3


-- 2.2. Pattern raised to a negative integer N:
-----------------------------------------------
-- A pattern raised to a negative integer results in a new pattern that matches
-- the original one AT MOST that many times. In particular, a pattern raised to
-- -1 means an OPTIONAL element.
local num = lpeg.S("+-")^-1 * lpeg.R("09")^1
```

```lua
print(num:match("+1234"))                              --> 6
print(num:match("-1234"))                              --> 6
print(num:match("1234"))                               --> 5
print(num:match("++1234"))                             --> nil
print(num:match("1"))                                  --> 2
print(num:match("+"))                                  --> nil


-- 2.3. Choices:
----------------
-- We can use the + operator to creates choices that are ORDERED and ALWAYS
-- POSSESSIVE, that is: if the first alternative matches, LPeg will not consider
-- the second one, independently of what comes next. Choices where the first
-- alternative is a prefix of the second always fail:
local p = (lpeg.P("a") + lpeg.P("ab")) * lpeg.P("c")
print(p:match("abc"))                                  --> nil

-- The order of the alternatives in LPeg is essential. Think of a machine trying
-- to match each pattern in turn:
--     p * q : if p succeeds, it proceeds to match q; if either of them fails,
--             the sequence fails
--     p + q : if p succeeds, it does not check q; if p fails, it check q; if
--             both fails, the sequence fails
--     e^0   : in a repetition, keeps trying to match e until that fails




-- 3. SIMPLE CAPTURES:
----------------------
-- Captures are patterns that produces values.

-- 3.1. Simple Capture:
-----------------------
-- The simplest capture in LPeg is created with the funciont lpeg.C, that
-- receives a pattern and returns a capture with the string that matched:
local p = loc.space^0 * lpeg.C(loc.alpha^1)  -- 0+ espaços seguidos por 1+ letra
print(p:match(" hello world"))                         --> hello
print(p:match("     helloworld"))                      --> helloworld

-- A pattern may have multiples captures, which can produce multiples values:
local p = lpeg.C(2) * lpeg.C(1)
print(p:match("hello"))                                --> he      l

-- Captures can be nested and, in that case, the outer captures come first:
local p = lpeg.C(lpeg.C(2) * 1 * lpeg.C(2))
print(p:match("hello"))                                --> hello     he     lo

-- CAUTION! The number of values produced by a pattern may depend on the
-- subject and the pattern (even if similar):

-- In this case, captures always capture something:
print(lpeg.C(lpeg.P("a")^0):match("aaa"))              --> aaa
print(lpeg.C(lpeg.P("a")^0):match(""))                 -->    (capture empty string)
-- But in this case, capture may capture not at all:
print((lpeg.C(lpeg.P("a"))^0):match("aaa"))            --> a     a     a
print((lpeg.C(lpeg.P("a"))^0):match(""))               --> 1 (no capture)
```

```lua
-- In this case, capture always capture something:
print(lpeg.C(lpeg.P"a"^-1):match("a"))             --> a
print(lpeg.C(lpeg.P"a"^-1):match(""))              -->   (capture empty string)
-- But in this case, capture may capture not at all:
print((lpeg.C(lpeg.P"a")^-1):match("a"))           --> a
print((lpeg.C(lpeg.P"a")^-1):match(""))            --> 1 (no capture)


-- 3.2. Position Capture:
-------------------------
-- lpeg.Cp creates a position capture, a pattern that captures the current
-- position in the subject where the match occurred:
local p = loc.space^0 * lpeg.Cp()
print(p:match("  hello"))                          --> 3
print(p:match("hello"))                            --> 1

local p = (loc.space^0 * lpeg.Cp() * loc.alpha^1)^0
print(p:match("hello my world"))                   --> 1     7     10




-- 4. PREDICATES:
-----------------
-- PEGs are always DETERMINISTIC, that is, there is only one way to match
-- a given pattern and subject. This allow us the NEGATION of a pattern with the
-- not predicate, an unary minus operator.

-- CAUTION: a not predicate never consumes any input and never produces any
-- captures!

local p = -lpeg.P("a")
print(p:match("bcd"))                              --> 1
print(p:match(""))                                 --> 1
print(p:match("abc"))                              --> nil

-- The pattern -lpeg.P(1) always match when there is no characters, that is, the
-- end of the subject. This is particularly useful to ensure that a successful
-- match consumed the entire subject:
local p = loc.digit^0 * -lpeg.P(1)
print(p:match("123"))                              --> 4
print(p:match("123 "))                             --> nil


-- 4.1. Searching:
------------------
-- (skiped for now)


-- 4.2. Identifiers in the real world:
--------------------------------------
-- (skiped for now)


-- 4.3. List of arguments:
--------------------------
-- (skiped for now)
```

```lua
-- 5. AGGREGATING CAPTURES:
--------------------------
-- When a match produces multiple captures, we'll want to aggregate those
-- results. For simple cases we can use Lua directly to do the aggregation,
-- but the most general is the FUNCTION CAPTURE, denoted by a division
-- operator. When matching an expression p/f, LPeg first matches p; then it
-- calls f passing as arguments all captures produced by p; f process this
-- values and the values returned by f become the final captures of the
-- whole expression.
function soma(a, b)
   return a + b
end

local num = loc.digit^1 / tonumber
local p = (num * "+" * num) / soma
print(p:match("2+3"))                              --> 5

-- CAUTION: LPeg gives no guarantees abou when and if the function in a
-- function capture will be called, so, these function shoul NOT produce
-- side effects.



-- 6. SIMPLE ARITHMETIC EXPRESSIONS:
-----------------------------------

-- 6.1. Additive operators:
---------------------------
S = loc.space^0
num = (loc.digit^1 / tonumber) * S
opA = lpeg.C("+") * S
opS = lpeg.C("-") * S
V = -lpeg.P(1)
exp = S * num * ((opA + opS) * num)^0 * V

print(exp:match(" 23 + 34 - 15 "))                 --> 23   +   34   -   15

-- To capture and process all values in a list, LPeg offers a special capture to
-- that end: TABLE CAPTURE, with the funciont lpeg.Ct:
exp = lpeg.Ct(S * num * ((opA + opS) * num)^0 * V)
print(pt.pt(exp:match("34 + 89 - 23")))

-- To process the values, we create a FUNCTION CAPTURE, generally called fold,
-- with the processing logic:
function fold(list)
   local acc = list[1]
   for i = 2, #list, 2 do
      if list[i] == "+" then
         acc = acc + list[i + 1]
      else
         acc = acc - list[i + 1]
      end
   end
   return acc
end

exp = lpeg.Ct(S * num * ((opA + opS) * num)^0 * V) / fold
print(exp:match(" 34 + 89 - 23 "))                 --> 100
```

```lua
-- To make the things more reusable, we use funcional programming and represent
-- the operators with a function that performs the operation, and modify the
-- fold function according:
function soma(a, b)
   return a + b
end

function subtrair(a, b)
   return a - b
end

function fold(list)
   local acc = list[1]
   for i = 2, #list, 2 do
      local op = list[i]          -- get operation
      acc = op(acc, list[i + 1]) -- apply operatrion
   end
   return acc
end

-- Now we need to use a CONSTANT CAPTURE with lpeg.Cc, that creates a capture
-- that produces a constante value without consumig any input, so we can
-- redefine the patterns that match operators:
opA = lpeg.P("+") * lpeg.Cc(soma) * S      -- opA produces the soma function
opS = lpeg.P("-") * lpeg.Cc(subtrair) * S  -- opS produces the subratir function

exp = lpeg.Ct(S * num * ((opA + opS) * num)^0 * V)
print(pt.pt(exp:match("34 + 89 - 23")))

exp = lpeg.Ct(S * num * ((opA + opS) * num)^0 * V) / fold
print(exp:match(" 34 + 89 - 23 "))                --> 100


-- 6.2. Multiplicative operators:
---------------------------------
-- We need two expressions for handling of hierarchy: the first expression, we
-- usually call "term" contains only multiplicative operators; the second
-- expressions, usually call "exp" contains only additive operators; This form
-- a form of SOP expression: Sum of Products.
local function somar(a, b)
   return a + b
end

local function subtrair(a, b)
   return a - b
end

local function multiplicar(a, b)
   return a * b
end

local function dividir(a, b)
   return a / b
end

local function resto(a, b)
   return a % b
end
```

```lua
local function fold(list)
   local acc = list[1]
   for i = 2, #list, 2 do
      local op = list[i]
      acc = op(acc, list[i + 1])
   end
   return acc
end

local S = loc.space^0
local V = -lpeg.P(1)
local sinal = lpeg.S("+-")^-1
local num = ((sinal * loc.digit^1) / tonumber) * S
local opA = lpeg.P("+") * lpeg.Cc(somar) * S
local opS = lpeg.P("-") * lpeg.Cc(subtrair) * S
local opM = lpeg.P("*") * lpeg.Cc(multiplicar) * S
local opD = lpeg.P("/") * lpeg.Cc(dividir) * S
local opR = lpeg.P("%") * lpeg.Cc(resto) *S

local term = lpeg.Ct(S * num * ((opM + opD + opR) * num)^0) / fold
local exp = lpeg.Ct(S * term * ((opA + opS) * term)^0 * V) / fold

print(exp:match("23 * 3 - 14 / 2"))              --> 62.0
print(exp:match("34 + 89 * 23"))                 --> 2081
print(exp:match("5 + -5 + 2 * -5"))              --> -10
print(exp:match("5 % 3"))                        --> 2
print(exp:match("5 % 3 * 4  "))                  --> 8
print(exp:match("5 % 3 * 4 abc"))                --> nil



-- 7. GRAMMAR:
--------------
-- From sections 1 to 6, we cover the "parsing expressions" part of our
-- formalism, Parsing Expression Grammar (PEG). In this section we'll see
-- the oter part, "grammar", wich permitt us to do more complex things,
-- like recursive patterns.

-- Informally, a grammar is a set of named patterns where each pattern can refer
-- to other patterns (or to itself) through their names. For example, a simple
-- Lisp grammar for S-expressions is:
--     sexp <- name / '(' list ')'
--     list <- (sexp spaces)*
-- A S-exp is a name OR a list enclosed in parenthesis. A list is zero or more
-- s-exp followed by spaces.

-- LPeg uses Lua tables to support grammars:
--     - Key: is the name of a patterns
--     - Value: is the corresponding LPeg pattern

-- To refer to a pattern, we use a NONTERMINAL (or variable) pattern, through
-- the lpeg.V function (receives the name of a pattern and returns the
-- corresponding nonterminal pattern).

-- As entries in a table have no intrinsic order, we need also a way to signal
-- which pattern is the initial one, by assigning the initial pattern itself or
-- it's name to the index 1 of the table.
```

```lua
-- Here is a grammar to S-exp:
local name = loc.alpha^1
local spaces = loc.space^0
local g = {
   [1] = "sexp",
   sexp = name + "(" * lpeg.V"list" * ")",
   list = (spaces * lpeg.V"sexp" * spaces)^0
}

-- Once we have the grammar in a table, we need do "close" the table and get
-- the final pattern with lpeg.P (this means to internally connect all
-- nonterminals to theis respective patterns, raising and error if there is
-- any undefined nonterminal):
local p = lpeg.P(g)
p = p * -lpeg.P(1)  -- ATTENTION: this is necessary to check the whole subject

print(p:match("(() (a (b c)))"))                    --> 50
print(p:match("(a (b) (c d))"))                     --> 14
print(p:match("(a (b) (c d)"))                      --> nil
print(p:match("a (b) (c d))"))                      --> nil

-- Finally, we can add captures to our pattern and use a simplier syntax:
local p = lpeg.P{"sexp",
                 sexp = lpeg.C(name) + "(" * lpeg.V"list" * ")",
                 list = lpeg.Ct((spaces * lpeg.V"sexp" * spaces)^0)
}
p = p * -lpeg.P(1)

print(pt.pt(p:match("(esta e (uma lista) (em) lisp)")))
```