```
-- Week 1, Activity 20:
-- Parenthesized Expressions

local lpeg = require "lpeg"
local pt = require "pt"

-- Lexical Elements:
vazio = -lpeg.P(1)
espaco = lpeg.S(" \n\t")^0
sinal = lpeg.S("+-")^-1
numero = ((sinal * lpeg.R("09")^1) / tonumber) * espaco
opAdd = lpeg.C(lpeg.S("+-")) * espaco
opMul = lpeg.C(lpeg.S("*/%")) * espaco
opExp = lpeg.C(lpeg.P("^")) * espaco

-- Fold function for exponentials
function foldexp(lst)
  local acc = lst[#lst]
  for i = #lst - 1, 2, -2 do
    acc = lst[i - 1] ^ acc
  end
  return acc
end

-- Fold function for multiplicative and additive operators
function fold(lst)
  local acc = lst[1]
  for i = 2, #lst, 2 do
    if lst[i] == "+" then
      acc = acc + lst[i + 1]
    elseif lst[i] == "-" then
      acc = acc - lst[i + 1]
    elseif lst[i] == "*" then
      acc = acc * lst[i + 1]
    elseif lst[i] == "/" then
      acc = acc / lst[i + 1]
    elseif lst[i] == "%" then
      acc = acc % lst[i + 1]
    elseif lst[i] == "^" then
      acc = acc ^ lst[i + 1]
    else
      error("unknown operator")
    end
  end
  return acc
end

-- Pattern:
local pot = espaco * lpeg.Ct(numero * (opExp * numero)^0) / foldexp
local term = espaco * lpeg.Ct(pot * (opMul * pot)^0) / fold
local sum = espaco * lpeg.Ct(term * (opAdd * term)^0) / fold * vazio

-- Some tests:
-- These must be OK:
local teste = "-10 + 2 ^ 2 ^ 3 + 4 + 50"
local tabela = espaco * lpeg.Ct(numero * ((opMul + opAdd + opExp) * numero)^0) * v
azio
print(teste)
print(pt.pt(tabela:match(teste)))
print(sum:match(teste))
```

```
-- Until here, everything is OK. But frequently we'll want to have parenthesis
-- in ours expressions, that is, we will have subexpressions inside our
-- expressions, delimited by parenthesis.
-- This kind of thing is not possible in LPEG directly with patterns, because
-- we'll need to use recursion: a subpression of a subexpression of... and so.
-- For this we'll need to use a grammar to define this recursive pattern.

-- Let's define the parenthesis:
local OP = lpeg.P("(") * espaco
local CP = lpeg.P(")") * espaco

-- We need to define a grammar (in this case called g) with our pattern and,
-- in each pattern, we'll use variables (lpeg.V) to refer to other patterns.
-- We also need to specify the first rule, that rule that will start the
-- recursion process. Note: we can not check for EOL inside the grammar,
-- because expressions can have another expression but, for this, we check
-- for EOL after the grammar.
g = lpeg.P{
  [1] = "exp",
  primary = espaco * numero + OP * lpeg.V"exp" * CP,
  pot = espaco * lpeg.Ct(lpeg.V"primary" * (opExp * lpeg.V"primary")^0) / foldexp,
  term = espaco * lpeg.Ct(lpeg.V"pot" * (opMul * lpeg.V"pot")^0) / fold,
  exp = espaco * lpeg.Ct(lpeg.V"term" * (opAdd * lpeg.V"term")^0) / fold
}
g = g * vazio

-- Let's test:
local teste = "2 * (2 + 4) * 10"
print(teste)
print(g:match(teste))

local teste = "2 ^ (2 * (5 - 1))"
print(teste)
print(g:match(teste))

-- To cleanup a bit our code, it's usual to define the non-terminals variables
-- before the grammar, and inside the grammar just use the names defined:
local exp = lpeg.V"exp"
local primary = lpeg.V"primary"
local pot = lpeg.V"pot"
local term = lpeg.V"term"

g = lpeg.P{"exp",
  primary = espaco * numero + OP * exp * CP,
  pot = espaco * lpeg.Ct(primary * (opExp * primary)^0) / foldexp,
  term = espaco * lpeg.Ct(pot * (opMul * pot)^0) / fold,
  exp = espaco * lpeg.Ct(term * (opAdd * term)^0) / fold
}
g = g * vazio

-- Let's test again:
local teste = "2 * (2 + 4) * 10"
print(teste)
print(g:match(teste))

local teste = "2 ^ (2 * (5 - 1))"
print(teste)
print(g:match(teste))
```