```lua
-- Week 3, Activity 06
-- Return statement
--
-- Student: Abrantes Araújo Silva Filho

local lpeg = require "lpeg"
local pt = require "pt"
local loc = lpeg.locale()



-------------------------------------------------------------------------
-- FRONTEND: PARSER
-- Our frontend is a parser that gets a source code as input and produces an
-- intermediate representation of the program (an AST)
-------------------------------------------------------------------------

------------------------- Initial patterns: ------------------------------

-- White spaces and "end of subject"
local spc = loc.space^0
local eos = -lpeg.P(1)

-- Basic identifiers
local alpha = lpeg.R("az", "AZ", "__")
local digit = loc.digit
local alphanum = alpha + digit
local dot = lpeg.P(".")
local ID = lpeg.C(alpha * alphanum^0) * spc

-- Numeric constants
local sign = lpeg.S("+-")
local hexpre = lpeg.P("0") * lpeg.S("Xx")
local hexdig = lpeg.R("AF", "af", "09")
local hexdec = hexpre * hexdig^1 * spc
local expSym = lpeg.S("Ee")
local sufEXP = (expSym * sign^-1 * digit^1)^-1 * spc
local decimal = ((digit^1 * dot^-1 * digit^0) + (dot^-1 * digit^1)) * sufEXP * spc

-- Punctuators:
local OP = lpeg.P("(") * spc
local CP = lpeg.P(")") * spc
local OB = lpeg.P("{") * spc
local CB = lpeg.P("}") * spc
local SC = lpeg.P(";") * spc

-- Numeric operators (binary and unary)
local opPot = lpeg.C(lpeg.P("^")) * spc
local opMul = lpeg.C(lpeg.S("*/%")) * spc
local opAdd = lpeg.C(lpeg.S("+-")) * spc
local opUnaMin = lpeg.P("-") * spc
local opUnaPlus = lpeg.P("+") * spc

-- Relational and equality operators
local lt = lpeg.C(lpeg.P("<")) * spc
local lte = lpeg.C(lpeg.P("<=")) * spc
local gt = lpeg.C(lpeg.P(">")) * spc
local gte = lpeg.C(lpeg.P(">=")) * spc
local eq = lpeg.C(lpeg.P("==")) * spc
local neq = lpeg.C(lpeg.P("!=")) * spc
local opRel = (lte + gte + lt + gt + eq + neq) * spc
```

```lua
-- Assignments for statements
local Assign = lpeg.P("=") * spc

-- Reserved words
local ret = lpeg.P("return") * spc

------------------------- Functions for the Parser: -------------------------

-- Function to get a number and return a node for an AST represeenting a number:
local function nodeNum(numero)
   return { tag = "numero",
            val = numero }
end

-- Function to get a variable and return a node for as AST
local function nodeVar(var)
   return { tag = "variable",
            var = var }
end

-- Function to get an assignment and return a node for AST
local function nodeAssign(id, exp)
   return { tag = "assign",
            id = id,
            exp = exp }
end

-- Function to get a statement and, optionally, a list of statements,
-- and return a note for AST:
local function nodeSeq(st1, st2)
   if st2 == nil then
      return st1
   else
      return { tag = "seq",
               st1 = st1,
               st2 = st2 }
   end
end

-- Function to treat an empty block:
local function nodeNull()
   return { tag = "nothing" }
end

-- Function to reat a return:
local function nodeRet(exp)
   return { tag = "ret", exp = exp }
end

-- Functions to fold a list and convert the list to an AST:
--   input: list: {n1, "+", n2, "+", n3, ...}
--   output: AST: {...{ op = "+", e1 = {op = "+", e1 = n1, e2 = n2}, e2 = n3}...}

-- foldBinEsq = operators with left-associativity
local function foldBinEsq(list)
   local tree = list[1]
   for i = 2, #list, 2 do
      tree = { tag = "binop",
               esq = tree,
```

```lua
                op = list[i],
                dir = list[i + 1] }
   end
   return tree
end

-- foldBinDir = operator with right-associativity
local function foldBinDir(list)
   local tree = list[#list]
   for i = #list - 1, 2, -2 do
      tree = { tag = "binop",
               esq = list[i - 1],
               op = list[i],
               dir = tree }
   end
   return tree
end

-- foldUnaMinus = unary minus
local function foldUnaMin(numero)
   return { tag = "menos_unario" , op = numero }
end

-- foldUnaPlus = unary plus
local function foldUnaPlus(numero)
   return { tag = "mais_unario", op = numero }
end

------------------- Our grammar for mathematic expression: --------------------

local numero = spc * ((hexdec / tonumber) + (decimal / tonumber)) / nodeNum * spc
local var = ID / nodeVar          -- variables
local stat = lpeg.V"stat"         -- a statement
local stats = lpeg.V"stats"       -- a list of statements
local block = lpeg.V"block"       -- a block of code
local primary = lpeg.V"primary"   -- primary (for recursion and parenthesis)
local pot = lpeg.V"pot"           -- exponentials
local unarymp = lpeg.V"unarymp"   -- unary minus or unary plus
local term = lpeg.V"term"         -- multiplicative expressions
local exp = lpeg.V"exp"           -- aditive expressions
local rel = lpeg.V"rel"           -- relational expressions

grammar = lpeg.P{"stats",
   stats = stat * (SC * stats)^-1 / nodeSeq,
   block = OB * stats * SC^-1 * CB
         + OB * SC^-1 * CB / nodeNull,
   stat = block
        + ID * Assign * rel / nodeAssign
        + ret * exp / nodeRet,
   primary = spc * numero + OP * rel * CP + var,
   pot = lpeg.Ct(spc * primary * (opPot * primary)^0) / foldBinDir,
   unarymp = (opUnaMin * unarymp / foldUnaMin) +
             (opUnaPlus * unarymp / foldUnaPlus) + pot,
   term = lpeg.Ct(spc * unarymp * (opMul * unarymp)^0) / foldBinEsq,
   exp = lpeg.Ct(spc * term * (opAdd * term)^0) / foldBinEsq,
   rel = lpeg.Ct(spc * exp * (opRel * exp)^0) / foldBinEsq
}
grammar = spc * grammar * eos

------------------------------- The parser per si: ---------------------------
```

```lua
local function parse(input)
    return grammar:match(input)
end


--------------------------------------------------------------------------------
-- BACKEND: CODE GENERATOR
-- Our backend is a code generator that get's an AST and generate the final
-- output of the compiler
--------------------------------------------------------------------------------
-- Function to add opcodes:
local function addCode(state, op)
    local code = state.code
    code[#code + 1] = op
end

-- Operators:
local ops = {["+"] = "add", ["-"] = "sub",
             ["*"] = "mul", ["/"] = "div", ["%"] = "rem",
             ["^"] = "exp",
             ["<="] = "lte", [">="] = "gte", ["=="] = "eq", ["!="] = "ne",
             [">"] = "gt", ["<"] = "lt"}

-- Function to specify the operations by type (tag) of node:
local function codeExp(state, ast)
    if ast.tag == "numero" then
        addCode(state, "push")
        addCode(state, ast.val)
    elseif ast.tag == "variable" then
        addCode(state, "load")
        addCode(state, ast.var)
    elseif ast.tag == "binop" then
        codeExp(state, ast.esq)
        codeExp(state, ast.dir)
        addCode(state, ops[ast.op])
    elseif ast.tag == "menos_unario" then
        codeExp(state, ast.op)
        addCode(state, "inverter")
    elseif ast.tag == "mais_unario" then
        codeExp(state, ast.op)
        addCode(state, "manter")
    else
        error("invalid tree")
    end
end

-- Function to assign expressions to variables:
local function codeStat(state, ast)
    if ast.tag == "assign" then
        codeExp(state, ast.exp)  -- evaluates the expression to get the value assign
ed
        addCode(state, "store")
        addCode(state, ast.id)
    elseif ast.tag == "seq" then
        codeStat(state, ast.st1)
        codeStat(state, ast.st2)
    elseif ast.tag == "nothing" then
        -- do nothing here
    elseif ast.tag == "ret" then
        codeExp(state, ast.exp)
```

```lua
            addCode(state, "ret")
        else
            error("invalid tree")
        end
    end
end


-- The compiler per si:
local function compile(ast)
    local state = { code = {} }
    codeStat(state, ast)
    -- next 3 lines forces the compiler to always end with a return:
    addCode(state, "push")
    addCode(state, 0)
    addCode(state, "ret")
    return state.code
end



-------------------------------------------------------------------------------
-- INTERPRETER
-- Receives the intermediate code produced by the compiler and empty stack and,
-- when finished, leaves the result of the expression on the top of the stack.
-------------------------------------------------------------------------------
-- The interpreter:
local function run(code, mem, stack)
    local pc = 1                        -- program counter
    local top = 0                       -- top of stack
    while true do
        -- In the block: code to help debug the interpreter, if needed
        --[[
        io.write("-->")
        for i = 1, top do io.write(stack[i], " ") end
        io.write("\n", code[pc], "\n")
        --]]
        if code[pc] == "ret" then
            return
        elseif code[pc] == "push" then
            pc = pc + 1
            top = top + 1
            stack[top] = code[pc]
        elseif code[pc] == "add" then
            stack[top - 1] = stack[top - 1] + stack[top]
            top = top - 1
        elseif code[pc] == "sub" then
            stack[top - 1] = stack[top - 1] - stack[top]
            top = top - 1
        elseif code[pc] == "mul" then
            stack[top - 1] = stack[top - 1] * stack[top]
            top = top - 1
        elseif code[pc] == "div" then
            stack[top - 1] = stack[top - 1] / stack[top]
            top = top - 1
        elseif code[pc] == "rem" then
            stack[top - 1] = stack[top - 1] % stack[top]
            top = top - 1
        elseif code[pc] == "exp" then
            stack[top - 1] = stack[top - 1] ^ stack[top]
            top = top - 1
        elseif code[pc] == "gte" then
            stack[top - 1] = (stack[top - 1] >= stack[top]) and 1 or 0
```

```lua
            top = top − 1
        elseif code[pc] == "lte" then
            stack[top − 1] = (stack[top − 1] <= stack[top]) and 1 or 0
            top = top − 1
        elseif code[pc] == "gt" then
            stack[top − 1] = (stack[top − 1] > stack[top]) and 1 or 0
            top = top − 1
        elseif code[pc] == "lt" then
            stack[top − 1] = (stack[top − 1] < stack[top]) and 1 or 0
            top = top − 1
        elseif code[pc] == "eq" then
            stack[top − 1] = (stack[top − 1] == stack[top]) and 1 or 0
            top = top − 1
        elseif code[pc] == "ne" then
            stack[top − 1] = (stack[top − 1] ~= stack[top]) and 1 or 0
            top = top − 1
        elseif code[pc] == "inverter" then
            stack[top] = −stack[top]
        elseif code[pc] == "manter" then
            -- do nothing
        elseif code[pc] == "load" then
            pc = pc + 1
            local id = code[pc]
            top = top + 1
            stack[top] = mem[id]
        elseif code[pc] == "store" then
            pc = pc + 1
            local id = code[pc]
            mem[id] = stack[top]
            top = top − 1
        else
            error("unknown instruction")
        end
        pc = pc + 1
    end
end



-------------------------------------------------------------------------------
-- RUN!
-- Let's read some source code and execute the interpreter!
-------------------------------------------------------------------------------

-- Get's the source code (only a number for now):
local input = io.read("a")

-- The frontend (parser) generates as AST:
local ast = parse(input)
print(pt.pt(ast))

-- The backend (code generator) compiles AST to intermediate code:
local code = compile(ast)
print(pt.pt(code))

-- We run the interpreter passing as arguments the
-- intermediate code, a memory for global variables, and the stack:
local stack = {}
local mem = {}
run(code, mem, stack)
print(stack[1])
```