```lua
-- Week 2, Activity 8:
-- Products

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 in an AST
--------------------------------------------------------------------------------
-- Initial patterns:
local spc = loc.space^0
local vazio = -lpeg.P(1)
local sinal = lpeg.S("+-")^-1
local hexdig = lpeg.R("AF", "af", "09")
local hexpre = lpeg.P("0") * lpeg.S("Xx")
local opAS = lpeg.C(lpeg.S("+-")) * spc
local opMD = lpeg.C(lpeg.S("*/%")) * spc

-- Function that get's a number and return a node of AST
function node(numero)
   return {
      tag = "numero",
      val = numero
   }
end

-- What is a number? Note that an AST node is returned
local decnum = ((sinal * loc.digit^1) / tonumber) / node * spc
local hexnum = ((sinal * hexpre * hexdig^1) / tonumber) / node * spc
local numero = spc * (hexnum + decnum) * spc

-- Function 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}...}
local function foldBin(list)
   local tree = list[1]
   for i = 2, #list, 2 do
      tree = { tag = "binop", e1 = tree, op = list[i], e2 = list[i + 1] }
   end
   return tree
end


-- Multiplications and Summations:
local term = lpeg.Ct(spc * numero * (opMD * numero)^0) / foldBin
local exp = lpeg.Ct(spc * term * (opAS * term)^0) / foldBin

-- The parser per si:
local function parse(input)
   return exp: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
```

```lua
--------------------------------------------------------------------------------
-- Function to add opcodes:
local function addCode(state, op)
   local code = state.code
   code[#code + 1] = op
end

local ops = {["+"] = "add", ["-"] = "sub",
             ["*"] = "mul", ["/"] = "div", ["%"] = "rem"}

-- 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 == "binop" then
      codeExp(state, ast.e1)
      codeExp(state, ast.e2)
      addCode(state, ops[ast.op])
   else
      error("invalid tree")
   end
end

-- The compiler per si:
local function compile(ast)
   local state = { code = {} }
   codeExp(state, ast)
   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, stack)
   local pc = 1                   -- program counter
   local top = 0                  -- top of stack
   while pc <= #code do
      if 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]
      else
         error("unknown instruction")
      end
      pc = pc + 1
   end
end


-- Tests:
-- 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 and AST:
local stack = {}
run(code, stack)
print(stack[1])
```