InClass.hs

{- Author: Richard Eisenberg, inspired by Steve Zdancewic
   File: Stack.hs
   Demonstrates ways of working with a stack. -}

{-# OPTIONS_GHC -W -Wno-unused-imports #-}

module Stack where

import Control.Monad ( forM_, when )

import Data.Vector       ( Vector )
import qualified Data.Vector as I   -- I for immutable
import Data.Vector.Mutable  ( IOVector )
import qualified Data.Vector.Mutable as M
import Data.IORef

-- This is based on the last part of Assignment 1.
-- In this version, we'll use a *functional* context. That is, a context
-- will be a function mapping strings to ints.
type Ctxt = String -> Int

emptyCtxt :: Ctxt
emptyCtxt v = error (v ++ " not found")

extendCtxt :: Ctxt -> String -> Int -> Ctxt
extendCtxt ctxt new_var new_val = \query -> if query == new_var then new_val else ctxt query

buildContext :: [(String, Int)] -> Ctxt
buildContext [] = emptyCtxt
buildContext ((var, val) : rest) = extendCtxt (buildContext rest) var val

-- One instruction in our stack machine
data Insn
  = IPushC Int        -- push an int64 constant onto the stack
  | IPushV String     -- push (lookup string ctxt) onto the stack
  | IMul              -- multiply the top two values on the stack
  | IAdd              -- add the top two values on the stack
  | INeg              -- negate the top value on the stack
  deriving (Eq, Show)

data Machine = M { stack :: IOVector Int -- in C: int[]
                      , spRef :: IORef Int    -- in C: int*
                      -- "stack pointer reference"
                      -- one past the last item on stack
                      , instructions :: Vector Insn
                      -- in C: const Insn[]
                      , pcRef :: IORef Int    -- in C: int*
                      -- "program counter reference"
                      }

newMachine :: Int -> Program -> IO Machine
newMachine size prog = do
  st <- M.new size
  sp <- newIORef 0   -- like malloc in C or "new" in Java
  let insns = I.fromList prog
  pc <- newIORef 0   -- programs start at the beginning
  pure (M { stack = st, spRef = sp
                      , instructions = insns, pcRef = pc })
InClass.hs

-- Print to stdout the current machine state (without the context)
printMachine :: Machine -> IO ()
printMachine (M { stack = st, spRef = sp, instructions = insns, pcRef = pc }) = do
    putStrLn "Stack:
    "
    putStrLn "  SP --> 
    "
    forM_ (reverse [0..sp_val-1]) $ \ stack_loc -> do
        stack_val <- M.read st stack_loc
        putStrLn ("         " ++ show stack_val)
    putStrLn "Instructions:
    "
    pc_val <- readIORef pc
    forM_ [0 .. I.length insns - 1] $ \ insn_loc -> do
        if (insn_loc == pc_val)
            then putStr "  PC --> 
            "
        else putStr "         
        let insn = insns I.! insn_loc
        putStrLn (show insn)
    putStrLn (" ++ show stack_val)"

-- access instruction where pc is pointing
step ctxt (M { stack = st, spRef = sp, instructions = insns, pcRef = pc }) = do
    pc_val <- readIORef pc
    let insn = insns I.! pc_val
    case insn of
        IPushC n -> push n
        IPushV v -> push (ctxt v)
        IMul -> do n1 <- pop
                  n2 <- pop
                  push (n1 * n2)
        IAdd -> do n1 <- pop
                  n2 <- pop
                  push (n1 + n2)
        INeg -> do n <- pop
                  push (-n)
    writeIORef pc (pc_val + 1)
    where
        push :: Int -> IO ()
        push n = do
            sp_val <- readIORef sp
            when (sp_val == M.length st)
                (error "stack not big enough")
            M.write st sp_val n
            writeIORef sp (sp_val + 1)

        writeIORef pc (pc_val + 1)
        -- *pc = pc_val + 1

        pop :: IO Int
        pop = do
            sp_val <- readIORef sp
            when (sp_val == 0)
                (error "stack underflow")
            let new_sp_val = sp_val - 1
            writeIORef sp new_sp_val
            M.read st new_sp_val

        pop = do
            sp_val <- readIORef sp
            when (sp_val == 0)
                (error "stack underflow")
            let new_sp_val = sp_val - 1
            writeIORef sp new_sp_val
            M.read st new_sp_val

{|}
InClass.hs

145:  -- Carry out one instruction, in a given context, producing an output stack.
146:  step :: Ctxt -> Stack -> Insn -> Stack
147:  step _ s             (IPushC n) = n : s
148:  step c s             (IPushV x) = c x : s
149:  step _ (v1 : v2 : s) IMul       = v1 * v2 : s
150:  step _ (v1 : v2 : s) IAdd       = v1 + v2 : s
151:  step _ (v : s)       INeg       = (-v) : s
152:  step _ _ _ = error "stack had too few values"
153:  
154:  -- Executing a program means repeatedly processing instructions
155:  execute :: Ctxt -> Stack -> Program -> Stack
156:  execute _ s []         = s  -- no more instructions to execute
157:  execute c s (i : cont) = execute c (step c s i) cont
158:  
159:  -- Extract the final, sole value from the stack. It must have 1 element.
160:  answer :: Stack -> Int
161:  answer [n] = n
162:  answer _   = error "no answer"
163:  
164:  -- Run a program in a given context for its variables
165:  run :: Ctxt -> Program -> Int
166:  run c p = answer (execute c [] p)
167:  
168:  -- Example:
169:  p1      = [IPushC 2, IPushC 3, IMul]
170:  answer1 = run empty_ctxt p1
171:  -}