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

-- An empty context maps all strings to error
emptyCtxt :: Ctxt
emptyCtxt v = error (v ++ " not found")

-- Extend a context with a new binding
extendCtxt :: Ctxt -> String -> Int -> Ctxt
extendCtxt ctxt new_var new_val
  = \query -> if query == new_var then new_val else ctxt query

-- Build a context from a list of (String, Int) pairs.
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)

-- A stack program is just a list of instructions.
type Program = [Insn]

data Machine = M { stack :: IOVector Int -- in C: int[
  , spRef :: IORef Int -- in C: int*  
                -- "stack pointer reference"
  , instructions :: Vector Insn
  , pcRef :: IORef Int -- in C: const Insn[]
  }

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

73: -- Print to stdout the current machine state (without the context)
74: printMachine :: Machine → IO ()
75: printMachine (M { stack = st,
76:     spRef = sp,
77:     instructions = insns,
78:     pcRef = pc }) = do
79:
80:   putStrLn "Stack:"
81:   sp_val <- readIORef sp    -- in C: sp_val = *sp;
82:   putStrLn "  SP --> "
83:   forM_ (reverse [0..sp_val-1]) $ \ stack_loc -> do
84:     stack_val <- M.read st stack_loc  -- stack_val = st[stack_loc]
85:     putStrLn ("         " ++ show stack_val)
86:   putStrLn "Instructions:"
87:   pc_val <- readIORef pc
88:   forM_ [0 .. I.length insns - 1] $ \ insn_loc -> do
89:     if (insn_loc == pc_val)
90:       then putStr "  PC --> "
91:       else putStr "         "
92:     let insn = insns !. insn_loc
93:     putStrLn (show insn)
94:   writeIORef pc (pc_val + 1)   -- *pc = pc_val + 1
95: where
96:   push :: Int → IO ()
97:   push n = do
98:     sp_val <- readIORef sp
99:     when (sp_val == M.length st)
100:       (error "stack not big enough")
101:     M.write st sp_val n
102:     writeIORef sp (sp_val + 1)
103: pop :: IO Int
104: pop = do
105:     sp_val <- readIORef sp
106:     when (sp_val == 0)
107:       (error "stack underflow") -- like Java’s throw
108:     let new_sp_val = sp_val - 1
109:     writeIORef sp new_sp_val
110: M.read st new_sp_val
111: {-
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 emptyCtxt p1
171: -}