{- Author: Richard Eisenberg, inspired by Steve Zdancewic
   File: Stack.hs

   Demonstrates ways of working with a stack, using mutable operations.
-}

{-# 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.

-- 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.
bUILD_CONTEXT :: [(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]

-- A stack machine has a data space for the stack, as well as the index of one
-- past the top of the stack (the bottom is always 0), the instructions,
-- and the program counter (PC), which tells us which instruction to run
-- next.
data Machine = M { stack        :: IOVector Int
                  , spRef        :: IORef Int
                  , instructions :: Vector Insn
                  , pcRef        :: IORef Int }

-- Create a new machine of the given size, ready to execute a given program
-- in the given context.
newMachine :: Int -> Program -> IO Machine
newMachine size prog = do
  st <- M.new size
  sp <- newIORef 0   -- SP starts at 0
  let insns = I.fromList prog
  pc <- newIORef 0   -- PC starts at 0
  pure (M { stack = st

StackIO.hs
StackIO.hs

73:    , spRef   = sp
74:    , instructions = insns
75:    , pcRef   = pc })
76:
77:    -- Print to stdout the current machine state (without the context)
78: printMachine :: Machine -> IO ()
79: printMachine (M { stack        = st
80:              , spRef        = sp
81:              , instructions = insns
82:              , pcRef        = pc }) = do
83:     putStrLn "Stack:
84:     sp_val <- readIORef sp
85:     putStrLn "  SP --> 
86:     forM_ (reverse [0..sp_val-1]) $ \ stack_loc -> do
87:         stack_val <- M.read st stack_loc
88:         putStrLn ("         "+ show stack_val)
89:     putStrLn "Instructions:
90:     pc_val <- readIORef pc
91:     forM_ [0 .. I.length insns - 1] $ \ insn_loc -> do
92:         if (insn_loc == pc_val)
93:             then putStrLn "  PC --> "
94:             else putStrLn "         "
95:         let insn = insns I.! insn_loc
96:         putStrLn (" " ++ show insn)
97:     putStrLn ""
98: step :: Ctxt -> Machine -> IO Bool
99: step ctxt (M { stack        = st
100:              , spRef        = sp
101:              , instructions = insns
102:              , pcRef        = pc }) = do
103:     -- Fetch the instruction
104:     pc_val <- readIORef pc
105:     let insn = insns I.! pc_val
106:     case insn of
107:         IPushC n -> push n
108:         IPushV x -> push (ctxt x)
109:         IMul     -> do n1 <- pop
110:             n2 <- pop
111:             push (n1 * n2)
112:         IAdd     -> do n1 <- pop
113:             n2 <- pop
114:             push (n1 + n2)
115:         INeg     -> do n <- pop
116:             push (-n)
117:     -- Increment the PC
118:     let new_pc_val = pc_val + 1
119:     writeIORef pc new_pc_val
120:     pure (new_pc_val == I.length insns)
121:     where
122:         push n = do
123:             sp_val <- readIORef sp
124:             when (sp_val == M.length st) $ 
125:                 error "Out of stack space"
126:             M.write st sp_val n
127:         pop = do
128:             sp_val <- readIORef sp
129:             when (sp_val == 0) $ 
130:                 error "stack underflow"
StackIO.hs

145: let new_sp_val = sp_val - 1
146: writeIORef sp new_sp_val
147: M.read st new_sp_val
148: -- Executing a machine means repeatedly processing instructions
149: execute :: Ctxt -> Machine -> IO ()
150: execute ctxt m = do
151:   done <- step ctxt m
152:   when (not done) $
153:     execute ctxt m
154: execute ctxt m
155: -- Extract the final, sole value from the machine. The stack must have 1 element.
156: answer :: Machine -> IO Int
157: answer (M { stack = st
158:           , spRef = sp }) = do
159: sp_val <- readIORef sp
160: when (sp_val /= 1) $
161:   error ("Stack has " ++ show sp_val ++ " values at end of run.")
162: M.read st 0
163: -- Run a program in a given context for its variables, with a given stack size
164: run :: Int -> Ctxt -> Program -> IO Int
165: run size ctxt prog = do
166: m <- newMachine size prog
167: execute ctxt m
168: answer m
169: -- Example:
170: p1 = [IPushC 2, IPushC 3, IMul]
171: answer1 = run 10 emptyCtxt p1
172: overflow = run 1 emptyCtxt p1
173: