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
   -- This creates a new function that checks if the query matches the new
   -- binding. If so, return the new value. Otherwise, look it up in the
   -- original context.

-- 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)

type Program = [Insn]

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

```
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 ""
90: putStrLn "Instructions:"
91: pc_val <- readIORef pc
92: forM_ [0 .. I.length insns - 1] $ \ insn_loc -> do
93: if (insn_loc == pc_val)
94: then putStrLn "  PC --> 
95: else putStrLn "         
96: let insn = insns I.! insn_loc
97: putStrLn (show insn)
98: putStrLn ""
99: putStrLn "-- Run the machine by one step in the given context.
100: -- Returns whether the machine is done running (True means "done")
101: step :: Ctxt -> Machine -> IO Bool
102: step ctxt (M { stack = st
103: , spRef = sp
104: , instructions = insns
105: , pcRef = pc }) = do
106: -- Fetch the instruction
107: pc_val <- readIORef pc
108: let insn = insns I.! pc_val
109: case insn of
110: IPushC n -> push n
111: IPushV x -> push (ctxt x)
112: IMul -> do n1 <- pop
113: n2 <- pop
114: push (n1 * n2)
115: IAdd -> do n1 <- pop
116: n2 <- pop
117: push (n1 + n2)
118: INeg -> do n <- pop
119: push (-n)
120: putStrLn "-- Increment the PC"
121: let new_pc_val = pc_val + 1
122: writeIORef pc new_pc_val
123: pure (new_pc_val == I.length insns)
124: where
125: push n = do
126: sp_val <- readIORef sp
127: when (sp_val == M.length st) $ error "Out of stack space"
128: M.write st sp_val n
129: modifyIORef sp (+1)
130: pop = do
131: sp_val <- readIORef sp
132: when (sp_val == 0) $ 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:
149: -- Executing a machine means repeatedly processing instructions
150: execute :: Ctxt -> Machine -> IO ()
151: execute ctxt m = do
152:   done <- step ctxt m
153:   when (not done) $
154:   execute ctxt m
155:
156: -- Extract the final, sole value from the machine. The stack must have 1 element.
157: answer :: Machine -> IO Int
158: answer (M { stack = st
159:           , spRef = sp }) = do
160:   sp_val <- readIORef sp
161:   when (sp_val /= 1) $
162:   error ("Stack has " ++ show sp_val ++ " values at end of run.")
163:
164: M.read st 0
165:
166: -- Run a program in a given context for its variables, with a given stack size
167: run :: Int -> Ctxt -> Program -> IO Int
168: run size ctxt prog = do
169:   m <- newMachine size prog
170:   execute ctxt m
171:   answer m
172:
173: -- Example:
174: p1 = [IPushC 2, IPushC 3, IMul]
175: answer1 = run 10 emptyCtxt p1
176: overflow = run 1 emptyCtxt p1
177: