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)

data Machine = M { stack        :: IOVector Int
                   , spRef        :: IORef Int
                   , instructions :: Vector Insn
                   , pcRef        :: IORef Int }

data Program = [Insn]

data Machine

-- 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
                 , spRef        = sp
                 , instructions = insns
                 , pcRef        = pc })
```
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:
84:   putStrLn "Stack:"
85:   sp_val <- readIORef sp
86:   putStrLn "  SP --> "
87:   forM_ (reverse [0..sp_val-1]) $ \ stack_loc -> do
88:     stack_val <- M.read st stack_loc
89:     putStrLn ("         " ++ show stack_val)
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 putStr "  PC --> "
95:       else putStr "         "
96:     insn = insns I.! insn_loc
97:   putStrLn (show insn)

102: -- Run the machine by one step in the given context.
103: -- Returns whether the machine is done running (True means "done")
104: step :: Ctxt -> Machine -> IO Bool
105: step ctxt (M { stack           = st
106:               , spRef          = sp
107:               , instructions = insns
108:               , pcRef          = pc }) = do
109:
110:   pc_val <- readIORef pc
111:   insn = insns I.! pc_val
112:   case insn of
113:     IPushC n -> push n
114:     IPushV x -> push (ctxt x)
115:     IMul     -> do n1 <- pop
116:                  n2 <- pop
117:                  push (n1 * n2)
118:     IAdd     -> do n1 <- pop
119:                  n2 <- pop
120:                  push (n1 + n2)
121:     INeg     -> do n <- pop
122:                  push (-n)
123:   new_pc_val = pc_val + 1
124:   writeIORef pc new_pc_val
125:   pure (new_pc_val == I.length insns)
126:   where
127:     push n = do
128:       sp_val <- readIORef sp
129:       when (sp_val == M.length st) $ error "Out of stack space"
130:     M.write st sp_val n
131:     modifyIORef sp (+1)
132:     pop = do
133:       sp_val <- readIORef sp
134:       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: -- 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: -- Extract the final, sole value from the machine. The stack must have 1 element.
155: answer :: Machine -> IO Int
156: answer (M { stack = st
157:     , spRef = sp }) = do
158:   sp_val <- readIORef sp
159:   when (sp_val /= 1) $
160:     error "$Stack has "$ ++ show sp_val ++ "$ values at end of run.$"
161:   M.read st 0
162: -- Run a program in a given context for its variables, with a given stack size
163: run :: Int -> Ctxt -> Program -> IO Int
164: run size ctxt prog = do
165:   m <- newMachine size prog
166:   execute ctxt m
167:   answer m
168: -- Example:
169: p1 = [IPushC 2, IPushC 3, IMul]
170: answer1 = run 10 emptyCtxt p1
171: overflow = run 1 emptyCtxt p1
172: