-- 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)

-- 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

-- This 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
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: putStrLn "  SP --> 
86: forM_ (reverse [0..sp_val-1]) $ \ stack_loc -> do
87: putStrLn "         " ++ show stack_val
88: putStrLn "Instructions:"
89: putStrLn "  PC --> 
90: forM_ [0 .. I.length insns - 1] $ \ insn_loc -> do
91: putStr "         
92: putStrLn (show insn)
93:
94: -- Run the machine by one step in the given context.
95: -- Returns whether the machine is done running (True means "done")
96: step :: Ctxt -> Machine -> IO Bool
97: step ctxt (M { stack = st
98: , spRef = sp
99: , instructions = insns
100: , pcRef = pc }) = do
101: -- Fetch the instruction
102: let insn = insns I.! pc_val
103: case insn of
104: IPushC n -> push n
105: IPushV x -> push (ctxt x)
106: IMul -> do n1 <- pop
107: n2 <- pop
108: push (n1 * n2)
109: IAdd -> do n1 <- pop
110: n2 <- pop
111: push (n1 + n2)
112: INeg -> do n <- pop
113: push (-n)
114: IMul -> do n1 <- pop
115: n2 <- pop
116: push (n1 * n2)
117: IAdd -> do n1 <- pop
118: n2 <- pop
119: push (n1 + n2)
120: INeg -> do n <- pop
121: push (-n)
122: IMul -> do n1 <- pop
123: n2 <- pop
124: push (n1 * n2)
125: IAdd -> do n1 <- pop
126: n2 <- pop
127: push (n1 + n2)
128: INeg -> do n <- pop
129: push (-n)
130: -- Increment the PC
131: let new_pc_val = pc_val + 1
132: writeIORef pc new_pc_val
133: pure (new_pc_val == I.length insns)
134: where
135: push n = do
136: when (sp_val == M.length st) $ 
137: error "Out of stack space"
138: M.write st sp_val n
139: modifyIORef sp (+1)
140: pop = do
141: when (sp_val == 0) $ 
142: error "stack underflow"
143: sp_val <- readIORef sp
144:
StackIO.hs

145: 146: let new_sp_val = sp_val - 1
147: writeIORef sp new_sp_val
148: M.read st new_sp_val
149:
150: -- Executing a machine means repeatedly processing instructions
151: execute :: Ctxt -> Machine -> IO ()
152: execute ctxt m = do
153:   done <- step ctxt m
154: when (not done) $]
155:   execute ctxt m
156:
157: -- Extract the final, sole value from the machine. The stack must have 1 element.
158: answer :: Machine -> IO Int
159: answer (M { stack = st
160: , spRef = sp }) = do
161: sp_val <- readIORef sp
162: when (sp_val /= 1) $]
163: error ("Stack has " ++ show sp_val ++ " values at end of run.")
164:
165: M.read st 0
166:
167: -- Run a program in a given context for its variables, with a given stack size
168: run :: Int -> Ctxt -> Program -> IO Int
169: run size ctxt prog = do
170: m <- newMachine size prog
171: execute ctxt m
172: answer m
173:
174: -- Example:
175: p1    = [IPushC 2, IPushC 3, IMul]
176: answer1 = run 10 emptyCtxt p1
177: overflow = run 1 emptyCtxt p1
178: