InClass.hs

1: {- Author: Richard Eisenberg, inspired by Steve Zdancewic
2: File: Stack.hs
3: Demonstrates ways of working with a stack.
4: -}
5: {-# OPTIONS_GHC -W -Wno-unused-imports #-}
6: module Stack where
7: import Control.Monad ( forM_, when )
8: import Data.Vector       ( Vector )
9: import qualified Data.Vector as I   -- I for immutable
10: import Data.Vector.Mutable  ( IOVector )
11: import qualified Data.Vector.Mutable as M
12: import Data.IORef
13: -- This is based on the last part of Assignment 1.
14: -- In this version, we’ll use a *functional* context. That is, a context
15: -- will be a function mapping strings to ints.
16: type Ctxt = String -> Int
17: emptyCtxt :: Ctxt
18: emptyCtxt v = error (v ++ " not found")
19: extendCtxt :: Ctxt -> String -> Int -> Ctxt
20: extendCtxt ctxt new_var new_val
21:   = \query -> if query == new_var then new_val else ctxt query
22:   -- This creates a new function that checks if the query matches the new
23:   -- binding. If so, return the new value. Otherwise, look it up in the
24:   -- original context.
25: buildContext :: [(String, Int)] -> Ctxt
26: buildContext []                  = emptyCtxt
27: buildContext ((var, val) : rest) = extendCtxt (buildContext rest) var val
28: data Insn
29:   = IPushC Int        -- push an int64 constant onto the stack
30:   | IPushV String     -- push (lookup string ctxt) onto the stack
31:   | IMul              -- multiply the top two values on the stack
32:   | IAdd              -- add the top two values on the stack
33:   | INeg              -- negate the top value on the stack
34: deriving (Eq, Show)
35: type Program = [Insn]
36: data Machine = M { stack :: IOVector Int -- in C: int[]
37:                  , spRef :: IORef Int    -- in C: int*
38:                             -- "stack pointer reference"
39:                  , instructions :: Vector Insn
40:                             -- one past the last item on stack
41:                  , pcRef :: IORef Int    -- in C: int*
42:                             -- "program counter reference"
43: } newMachine :: Int -> Program -> IO Machine
44: newMachine size prog = do
45:   st <- M.new size
46:   sp <- newIORef 0    -- like malloc in C or "new" in Java
47:   let insns = I.fromList prog
48:   pc <- newIORef 0    -- programs start at the beginning
49:   pure (M { stack = st, spRef = sp
50:       , 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:   putStrLn "Stack:"
80:   sp_val <- readIORef sp    -- in C: sp_val = *sp;
81:             -- in Java: sp_val = sp.value;
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    -- in C: pc_val = *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 I.! insn_loc
93:   putStrLn (" " ++ show stack_val)
94:   putStrLn ""
95:   putStrLn "Instructions:"
96:   pc_val <- readIORef pc
97:   forM_ [0 .. I.length insns - 1] $ \ insn_loc -> do
98:     if (insn_loc == pc_val)
99:       then putStr "  PC --> 
100:     else putStr "         
101:   let insn = insns I.! insn_loc
102:   putStrLn (" " ++ show stack_val)
103:   putStrLn ""
-- Carry out one instruction, in a given context, producing an output stack.

step :: Ctxt -> Stack -> Insn -> Stack
step _ s (IPushC n) = n : s
step c s (IPushV x) = c x : s
step _ (v1 : v2 : s) IMul = v1 * v2 : s
step _ (v1 : v2 : s) IAdd = v1 + v2 : s
step _ (v : s) INeg = (-v) : s

-- Executing a program means repeatedly processing instructions
execute :: Ctxt -> Stack -> Program -> Stack
execute _ s [] = s -- no more instructions to execute
execute c s (i : cont) = execute c (step c s i) cont

-- Extract the final, sole value from the stack. It must have 1 element.
answer :: Stack -> Int
answer [n] = n
answer _ = error "no answer"

-- Run a program in a given context for its variables
run :: Ctxt -> Program -> Int
run c p = answer (execute c [] p)

-- Example:
p1 = [IPushC 2, IPushC 3, IMul]
answer1 = run emptyCtx p1
