Demonstrates ways of working with a stack.

-- 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

-- Extend a context with a new binding

-- Build a context from a list of (String, Int) pairs.

-- 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

-- A stack program is just a list of instructions.

data Program = [Insn]

data Machine = M { stack :: IOVector Int -- in C: int[
  , spRef :: IORef Int -- in C: int*
    -- "stack pointer reference"
  , instructions :: Vector Insn
    -- one past the last item on stack
  , pcRef :: IORef Int -- in C: int*
    -- "program counter reference"
  }
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:   putStrLn "  SP --> 
82:   forM_ (reverse [0..sp_val-1]) $ \ stack_loc -> do
83:     stack_val <- M.read st stack_loc  -- stack_val = st[stack_loc]
84:     putStrLn ("         " ++ show stack_val)
85:   putStrLn "Instructions:
86:   pc_val <- readIORef pc
87:   forM_ [0 .. I.length insns - 1] $ \ insn_loc -> do
88:     if (insn_loc == pc_val)
89:       then putStr "  PC --> "
90:     else putStr "         "
91:     let insn = insns !. insn_loc
92:     putStrLn (show insn)
93:   putStrLn ("" ++ show stack_val)
94:   putStrLn ""

95: step :: Ctxt -> Machine -> IO ()
96: step ctxt (M { stack = st
97:              , spRef = sp
98:              , instructions = insns
99:              , pcRef = pc }) = do
100:   -- access instruction where pc is pointing
101:   pc_val <- readIORef pc
102:   let insn = insns !. pc_val
103:   case insn of
104:     IPushC n -> push n
105:     IPushV v -> push (ctxt v)
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:   writeIORef pc (pc_val + 1)   -- *pc = pc_val + 1
115:   putStrLn (show insn)
116:   putStrLn ""
117:   putStrLn "Instructions:"
InClass.hs

145: -- Carry out one instruction, in a given context, producing an output stack.
146: step :: Ctxt -> Stack -> Insn -> Stack
147: step _ s             (IPushC n) = n : s
148: step c s             (IPushV x) = c x : s
149: step _ (v1 : v2 : s) IMul       = v1 * v2 : s
150: step _ (v1 : v2 : s) IAdd       = v1 + v2 : s
151: step _ (v : s)       INeg       = (-v) : s
152: step _ _ _ = error "stack had too few values"

153:
154: -- Executing a program means repeatedly processing instructions
155: execute :: Ctxt -> Stack -> Program -> Stack
156: execute _ s []         = s  -- no more instructions to execute
157: execute c s (i : cont) = execute c (step c s i) cont
158:
159: -- Extract the final, sole value from the stack. It must have 1 element.
160: answer :: Stack -> Int
161: answer [n] = n
162: answer _   = error "no answer"

163:
164: -- Run a program in a given context for its variables
165: run :: Ctxt -> Program -> Int
166: run c p = answer (execute c [] p)
167:
168: -- Example:
169: p1     = [IPushC 2, IPushC 3, IMul]
170: answer1 = run emptyCtxt p1
171: -}