AP-Assignment 2

September 2022

kxs806

qmc887

Design and implementation

Answer to *** questions:

To define a monad it needs to have return and bind function explicitly declared. Comp monad use "return" function to receive "a" and put it into a Comp monad box like:(Right a,[]). Bind function(>>=) receive a monad m and deal with it with function f in the current environment and then return a new monad box b.If the calculate results an error we need to use abort function to return the error message and also keep the rest of strings in the list. When running the computation, if the first computation returns soem result, then we willl run the second computation, and f will be applied to the result of the first computation, like:(runComp (f a) e). During this, the environment will be passed and unchanged, because we only have read access. Again we can abort with an error and second computation, when there's an error occurs.

About the list comprehension part, we first check the clause list, if the list is empty we just eval the body and return it as a ListVal.Then the CCIf part we first get the expression need to be checked and then check it with truthy.If the result is True then eval the body and return it as a ListVal.If the result's False we just return an empty list. The CCFor part first eval the value of for variable. Then if it's a listVal we map it with withBinding function to calculate the exp with the variable equals to each of the elements in the list. Here if there are multiple calauses we want to eval them recursively from back to front, but the result consists of a lot of redundant ListVal(), that may be fixed by some way but we failed to find it, so we leave the multiple calauses as the comment. We will learn more adn try to fix it in the future.

Function implementation

Following are some implementations of functions we think important.

Monad operation function

abort: There are three kinds of RunError so we just receive them separately and wrap them in Comp.

look: We use the build-in lookup function to search for the vname in current env, if it exists, then wrap the value of the vname into Comp in shape of Right. Otherwise will get a EBadVar error.

withBinding: we first generate a new env with the new variables inserted in. and then use runComp to get the value from the monad box and then calculate it in the new env.As the new pair of vname:value is inserted in the head of new env, it will cover the old one(if have).

Helper function for interpreter

operate: We use patten match to get the origin number of operate number and calculate them. If there's no IntVal it will return an type error separately showing different operator needs Int value.

range: We split a range to get a value list of the range result to make apply function much easier to implement and to read. Inside the function there are two situations, one is v3 > 0 and v1 < v2, like (0,10,1), we concat current v1 with the result of the range from v1 + v3,v2. The other is v3 < 0 and v1 > v2, like (10,0,-1), we concat current v1 with the result of the range from v1 - v3, v2. For those not in these two situations we just return empty list as the stop condition of the resursion.

getexpstring, stringfylist,printfunc : these are help functions for print. Getexpstring can make some expression into string, stringfylist can help to make the value list into a string using recursive. printfunc uses getexpstring to make the value list into a string.

Assessment

Completeness

We completed most of the tasks in the assignment.

Correctness

We did a lot of tests and also on onlineTA, except for multiple calauses in CCFor, we passed most of other tests.

Efficiency

We used patten match a lot and we think it may not only make the code much more easy to read but also fasten the speed of the code.

Robustness

We considered most of the cases in the code and it will return a Left value in case of fail except of throw an error, that will make the code more robust and easier to read and use.

Maintainability

We made the code line as short as possible to make it readable.Also we add some comments on the functions we declared to make it easier to understand.We also split some very commenly used modult into functions so that the code become easier to read and change.

Appendix

```
-- Skeleton file for Boa Interpreter. Edit only definitions with
'undefined'

module BoaInterp
  (Env, RunError(..), Comp(..),
   abort, look, withBinding, output,
   truthy, operate, apply,
   eval, exec, execute)
  where

import BoaAST
import Control.Monad

type Env = [(VName, Value)]

data RunError = EBadVar VName | EBadFun FName | EBadArg String
  deriving (Eq, Show)

newtype Comp a = Comp {runComp :: Env -> (Either RunError a,
[String]) }

instance Monad Comp where
  return a = Comp(const (Right a, []))
  m >>= f = Comp(\e -> case runComp m e of
    (Left a,list) -> (Left a,list)
    (Right a,list) -> case runComp (f a) e of
      (Left a,s) -> (Left a,list <> s)
      (Right a,s) -> (Right a,list <> s)
    )

-- You shouldn't need to modify these
instance Functor Comp where
  fmap = liftM
instance Applicative Comp where
  pure = return; (<*>) = ap

-- Operations of the monad
```

```haskell
abort :: RunError -> Comp a
abort (EBadVar a) = Comp(const (Left (EBadVar a), []))
abort (EBadFun a) = Comp(const (Left (EBadFun a), []))
abort (EBadArg a) = Comp(const (Left (EBadArg a), []))

look :: VName -> Comp Value
look a = Comp(\e -> case lookup a e of
  Just x -> (Right x,[])
  Nothing -> (Left (EBadVar a),[])
  )

withBinding :: VName -> Value -> Comp a -> Comp a
withBinding x v m = Comp(\e -> let ne = (x,v):e
                              in runComp m ne)

output :: String -> Comp ()
output s = Comp(const (Right (), [s]))

-- Helper functions for interpreter
truthy :: Value -> Bool
truthy NoneVal = False
truthy FalseVal = False
truthy (IntVal 0) = False
truthy (StringVal "") = False
truthy (ListVal []) = False
truthy _ = True

operate :: Op -> Value -> Value -> Either String Value
operate Plus (IntVal v1) (IntVal v2) = Right (IntVal(v1+v2))
operate Plus _ _ = Left "can only plus integer velue"
operate Minus (IntVal v1) (IntVal v2) = Right (IntVal (v1 - v2))
operate Minus _ _ = Left "can only minus integer value"
operate Times (IntVal v1) (IntVal v2) = Right (IntVal (v1 * v2))
operate Times _ _ = Left "can only time integer value"
operate Div (IntVal v1) (IntVal v2)=
  if v2 == 0
    then Left "Divide by zero"
    else Right (IntVal (v1 `div` v2))
operate Div _ _ = Left "can only div integer value"
operate Mod (IntVal v1) (IntVal v2)=
  if v2 == 0
    then Left "Can't Mod by 0"
    else Right (IntVal (v1 `mod` v2))
operate Mod _ _ = Left "can only mod integer value"
operate Eq v1 v2 =
  if v1 == v2
    then Right TrueVal
    else Right FalseVal
operate Less (IntVal v1) (IntVal v2)=
  if v1 < v2
    then Right TrueVal
    else Right FalseVal
operate Less _ _ = Left "can only compare integer value"
operate Greater (IntVal v1) (IntVal v2) =
  if v1 > v2
    then Right TrueVal
    else Right FalseVal
operate Greater _ _ = Left "can only compare integer value"
operate In v1 (ListVal v2) =
  if v1 `elem` v2
    then Right TrueVal
    else Right FalseVal
operate _ _ _ = Left "undefined Operation"

range :: Value -> Value -> Value -> [Value]
range (IntVal v1) (IntVal v2) (IntVal v3)
  | v3 > 0 && v1 < v2 = IntVal v1 : range (IntVal (v1 + v3)) (IntVal
v2) (IntVal v3)
  | v3 < 0 && v1 > v2 = IntVal v1 : range (IntVal (v1 + v3)) (IntVal
v2) (IntVal v3)
  | otherwise = []
range _ _ _ = []



-- get the string of a value
getexpstring :: Value -> String
getexpstring NoneVal = "None"
getexpstring TrueVal = "True"
getexpstring FalseVal = "False"
getexpstring (IntVal x) = show x
getexpstring (StringVal s) = s
getexpstring (ListVal x) = "[" ++ stringfylist x ++ "]"

-- help to make list become a string
stringfylist :: [Value] -> String
stringfylist [] = ""
stringfylist [x] = getexpstring x
stringfylist (x:xs) = getexpstring x ++ ", " ++ stringfylist xs

-- print the value list
printFunc :: [Value] -> String
printFunc [] = ""
printFunc [x] = getexpstring x
printFunc (x:xs) = getexpstring x ++ " " ++ printFunc xs

apply :: FName -> [Value] -> Comp Value
apply "range" [v1,v2,NoneVal] = abort (EBadArg "bad arg")
apply "range" [v1,v2,v3] = let tmp = range v1 v2 v3
  in return (ListVal tmp)
apply "range" [v1] = apply "range" [IntVal 0,v1,IntVal 1]
apply "range" [v1,v2] = apply "range" [v1,v2,IntVal 1]
apply "range" _ = abort (EBadArg "range can only deal <= 3 paras")
apply "print" s = do
  output (printFunc s)
  return NoneVal
apply a _ = abort (EBadFun a)

-- Main functions of interpreter
eval :: Exp -> Comp Value
eval (Const v) = return v
eval (Var v) = look v
eval (Oper op e1 e2) = do
  v1 <- eval e1
  v2 <- eval e2
  case operate op v1 v2 of
    Right x -> return x
    Left x -> abort (EBadArg x)
```

```haskell
eval (Not v) = do
  tmp <- eval v
  if truthy tmp then return FalseVal else return TrueVal
eval (Call f es) = do
  vs <- eval (List es)
  case vs of
    (ListVal x) -> apply f x
    _ -> abort (EBadArg "call function err")
eval (List []) = return (ListVal [])
eval (List (e:es)) = do
  x <- eval e
  xs <- eval (List es)
  case xs of
    ListVal a -> return (ListVal (x : a))
    _ -> abort(EBadArg "List error")
eval (Compr e []) = do
  x <- eval e
  return (ListVal [x])
eval (Compr e ((CCFor v q):qs)) = do
    v1 <- eval q
    case v1 of
      ListVal l -> do
        -- l' <- mapM (\x -> withBinding v x (eval (Compr e qs))) l
        -- return (ListVal l')
        l' <- mapM (\x -> withBinding v x (eval e)) l
        return (ListVal l')
      _ -> abort (EBadArg "CCFor error,not a list")
eval (Compr e ((CCIf a):_)) = do
    check <- eval a
    let b = truthy check
    if b
      then do res<-eval e
              return (ListVal [res])
    else return (ListVal [])

exec :: Program -> Comp ()
exec [] = return ()
exec ((SDef v e):ps) = do
  x <- eval e
  withBinding v x (exec ps)
exec ((SExp e):ps) = do
  eval e
  exec ps

execute :: Program -> ([String], Maybe RunError)
execute p =  case runComp (exec p) [] of
  (Right (), s) -> (s, Nothing)
  (Left err, s) -> (s, Just err)
```

```haskell
-- Skeleton test suite using Tasty.
-- Fell free to modify or replace anything in this file

import BoaAST
import BoaInterp

import Test.Tasty
import Test.Tasty.HUnit

main :: IO ()
main = defaultMain $ localOption (mkTimeout 1000000) tests

tests :: TestTree
tests = testGroup "tests"[
   testGroup "test1"
   [testCase "check x EBadVar env empty" $
    runComp (look "x") [] @?= (Left (EBadVar "x"),[]),
  testCase "check x NoneVal env x" $
    runComp (look "x") [("x",NoneVal)] @?= (Right NoneVal,[]),
  testCase "check x IntVal env x" $
    runComp (look "x") [("x",IntVal 3)] @?= (Right (IntVal 3),[]),
  testCase "check x FalseVal env x" $
    runComp (look "x") [("x",FalseVal)] @?= (Right FalseVal,[]),
  testCase "check x NoneVal" $
    runComp (look "x") [("x",StringVal "text")] @?= (Right (StringVal
"text"),[]),
  testCase "check x StringVal env x" $
```

```
    runComp (look "x") [("x",ListVal [StringVal "text"])] @?= (Right
(ListVal [StringVal "text"]),[]),
  testCase "check x ListVal [StringVal]" $
    runComp (look "x") [("y",TrueVal),("w",FalseVal),("s",IntVal 3),
("x",ListVal [StringVal "text"])] @?= (Right (ListVal [StringVal
"text"]),[]),
  testCase "check x EBadVar env list of not x vars" $
    runComp (look "x") [("y",TrueVal),("w",FalseVal),("s",IntVal 3),
("x1",ListVal [StringVal "text"])] @?= (Left (EBadVar "x"),[])
   ],
   testGroup "test2"
   [testCase "withBinding NoneVal x" $
    runComp (withBinding "x" NoneVal (look "x")) [] @?= (Right
NoneVal,[]),
  testCase "withBinding IntVal x" $
    runComp (withBinding "x" (IntVal 3) (look "x")) [] @?= (Right
(IntVal 3),[]),
  testCase "withBinding TrueVal x" $
    runComp (withBinding "x" (TrueVal) (look "x")) [] @?= (Right
TrueVal,[]),
  testCase "withBinding FalseVal x" $
    runComp (withBinding "x" (FalseVal) (look "x")) [] @?= (Right
FalseVal,[]),
  testCase "withBinding StringVal x" $
    runComp (withBinding "x" (StringVal "text") (look "x")) [] @?=
(Right (StringVal "text"),[]),
  testCase "withBinding ListVal StringVal x" $
    runComp (withBinding "x" (ListVal [StringVal "text"]) (look "x"))
[] @?= (Right (ListVal [StringVal "text"]),[]),
  testCase "withBinding ListVal Vals x" $
    runComp (withBinding "x" (ListVal [StringVal "text",TrueVal,
FalseVal]) (look "x")) [] @?= (Right (ListVal [StringVal
"text",TrueVal, FalseVal]),[])],
   testGroup "output"
   [
      testCase "output empty string" $
      runComp (output "") [] @?= (Right (),[""]),
      testCase "output text" $
      runComp (output "asdasdasdasd") [] @?= (Right (),
["asdasdasdasd"])
    ],
   testGroup "Truthy"
   [testCase "truthy FalseVal" $
    truthy FalseVal @?= False,
  testCase "truthy (IntVal 10)" $
    truthy (IntVal 10) @?= True,
  testCase "truthy TrueVal" $
    truthy TrueVal @?= True,
  testCase "truthy (IntVal 3)" $
    truthy (IntVal 3) @?= True,
  testCase "truthy (ListVal [NoneVal])" $
    truthy (ListVal [NoneVal]) @?= True],
   testGroup "operate"
   [testCase "operate Plus" $
    operate Plus (IntVal 0) (IntVal 1) @?= Right (IntVal 1),
  testCase "operate Mod by Zero" $
    operate Mod (IntVal 0) (IntVal 0) @?= Left "Can't Mod by 0",
  testCase "test operate Eq" $
    operate Eq (IntVal 1) (IntVal 1) @?= Right TrueVal,
  testCase "operate Less 0<0" $
    operate Less (IntVal 0) (IntVal 0) @?= Right FalseVal,
  testCase "operate Greater 0>0" $
    operate Greater (IntVal 0) (IntVal 0) @?= Right FalseVal,
  testCase "operate In by []" $
    operate In (IntVal 0) (ListVal []) @?= Right FalseVal,
  testCase "operate Plus by IntVal" $
    operate Plus (IntVal 30) (IntVal 10) @?= Right (IntVal 40),
  testCase "operate Minus Negative" $
    operate Minus (IntVal 0) (IntVal 3) @?= Right (IntVal (-3)),
  testCase "operate Times by 10" $
    operate Times (IntVal 1) (IntVal 10) @?= Right (IntVal 10),
  testCase "operate Div by 4" $
    operate Div (IntVal 0) (IntVal 4) @?= Right (IntVal 0),
  testCase "operate Eq by Different Values" $
    operate Eq (IntVal 0) (TrueVal) @?= Right FalseVal,
  testCase "operate Eq StringVal" $
```

```
    operate Eq (StringVal "test") (StringVal "test") @?= Right
TrueVal,
  testCase "operate Eq []@?=4" $
    operate Eq (ListVal [IntVal 0]) (IntVal 4) @?= Right FalseVal,
  testCase "operate Less 4<0" $
    operate Less (IntVal 4) (IntVal 0) @?= Right FalseVal,
  testCase "operate Greater 4>0" $
    operate Greater (IntVal 4) (IntVal 0) @?= Right TrueVal,
  testGroup "eval"
    [testCase "eval (1+5+8)" $
    runComp (eval (Oper Plus (Oper Plus (Const (IntVal 1)) (Const
(IntVal 5))) (Const (IntVal 8)))) [] @?= (Right (IntVal 14),[]),
  testCase "eval not x (x not in env)" $
    runComp (eval (Not (Var "x"))) [] @?= (Left (EBadVar "x"),[]),
  testCase "eval not 1" $
    runComp (eval (Not (Const (IntVal 1)))) [] @?= (Right FalseVal,
[]),
  testCase "eval not 23" $
    runComp (eval (Not (Const (IntVal 23)))) [] @?= (Right FalseVal,
[]),
  testCase "eval not empty" $
    runComp (eval (Not (Const (StringVal "")))) [] @?= (Right
TrueVal,[]),
  testCase "eval not text" $
    runComp (eval (Not (Const (StringVal "text")))) [] @?= (Right
FalseVal,[]),
  testCase "eval not NoneVal" $
    runComp (eval (Not (Const NoneVal))) [] @?= (Right TrueVal,[]),
  testCase "eval Not x Trueval in env" $
    runComp (eval (Not (Var "x"))) [("x",TrueVal)] @?= (Right
FalseVal,[]) ,
  testCase "eval Not x Trueval in env" $
    runComp (eval (Not (List []))) [] @?= (Right TrueVal,[]),
  testCase "eval Call range(1,x) x unbound" $
    runComp (eval (Call "range" [Const (IntVal 1), Var "x"])) [] @?=
(Left (EBadVar "x"),[]),
  testCase "eval Call range(1,x) x bound 4" $
```

```
    runComp (eval (Call "range" [Const (IntVal 1), Var "x"]))
[("x",IntVal 4)] @?= (Right (ListVal [IntVal 1,IntVal 2,IntVal 3]),
[]),
  testCase "eval Call range(5)" $
    runComp (eval (Call "range" [Const (IntVal 4)])) [] @?= (Right
(ListVal [IntVal 0,IntVal 1,IntVal 2,IntVal 3]),[]),
  testCase "eval Call range(1,8,2)" $
    runComp (eval (Call "range" [Const (IntVal 1), Const (IntVal 8),
Const (IntVal 2)])) [] @?= (Right (ListVal [IntVal 1,IntVal 3,IntVal
5,IntVal 7]),[]),
  testCase "eval Call range(1,8)" $
    runComp (eval (Call "range" [Const (IntVal 1), Const (IntVal
8)])) [] @?= (Right (ListVal [IntVal 1,IntVal 2,IntVal 3,IntVal
4,IntVal 5,IntVal 6,IntVal 7]),[]),
  testCase "eval List (NoneVal, 2)" $
    runComp (eval (List [Const NoneVal, Const (IntVal 2)])) [] @?=
(Right (ListVal [NoneVal,IntVal 2]),[]),
  testCase "eval List (NoneVal)" $
    runComp (eval (List [Const NoneVal])) [] @?= (Right (ListVal
[NoneVal]),[]),
  testCase "eval List []" $
    runComp (eval (List [])) [] @?= (Right (ListVal []),[]),
  testCase "eval List (4,1,NoneVal, string)" $
    runComp (eval (List [Const (IntVal 4),Const (IntVal 1), Const
NoneVal,Const (StringVal "1")])) [] @?= (Right (ListVal [IntVal
4,IntVal 1,NoneVal,StringVal "1"]),[]),
  testCase "eval Compr x" $
    runComp (eval (Compr (Var "x") [])) [("x",IntVal 4)] @?= (Right
(ListVal [IntVal 4]),[]),
  testCase "eval Compr EBadVar" $
    runComp (eval (Compr (Var "x") [])) [] @?= (Left (EBadVar "x"),
[]),
  testCase "eval Compr (x+2) for range (1,5)" $
    runComp (eval (Compr (Oper Plus (Var "x") (Const (IntVal 2)))
[CCFor "x" (Call "range" [Const (IntVal 1),Const (IntVal 5)])]))
[("x",IntVal 4)] @?= (Right (ListVal [IntVal 3,IntVal 4,IntVal
5,IntVal 6]),[])],
   testGroup "exec Tests"
     [testCase "exec []" $
    runComp (exec []) [] @?= (Right (),[]),
  testCase "exec [print(x)]" $
    runComp (exec [SDef "x" (Call "print" [(Var "x")])]) [("x",IntVal
4)] @?= (Right (),["4"])]
  ]
  ]
```