Advanced Programming
Assignment 2
BOA Interpreter

GRC196 SZW935

September 22, 2021

1 Design and Implementation

1.1 `return` and `>>=`

In the instance of `Monad Comp` and with the use of the `Either` monad the `return` makes use of the type `Comp` and returns the `Right` right values of a and `mempty`. We can use `mempty` since the neutral element of a list of strings is an empty list of strings. For `>>=` we simply made use of the `case of` syntax and used `Left` if an error and `Right` for right values. In the second part of the `Warmup.hs` we were given the hint of exploiting the fact that the monad took from `Either` monad. This meant that we could make `>>=` much simple/abstract and not care about dealing with errors as the `Either` monad would be dealing with it. But found out that the return types were not compatible with that idea.

1.2 `print`

We have chosen to explain why and what we did to make print work, since it was not obvious to as how to implement it. To accommodate the functionalities we separated from `print` and made separate functions, i.e. `getSpace`, `convertList`, `getComma`. And another two functions `convertString` and `convertListElement` which call on the previous mentioned three functions. In this way we ensured that a comma would be placed only when print within a list.

2 Completeness

For this assignment we managed to implement all but the list comprehension. For the list comprehension we were thinking to use `case of` to check the input given to given to `Compr Exp [CClause]` where we would treat `CClause` as a list, such that, we get `(c:cs)` where we first match for `CCFor`, where we would then evaluate its expression and then again use `case of` to check for values. Here we were thinking that our `withBinding` would be called and then here it would

either return the values produced or an error. Then we would match on `CCIf` where we evaluate its expression and then check whether the results is true or false.

3 Correctness

Our testing tells us that, at least with the functionality that we implemented, it works as intended without bugs. Our test suite comprises of numerous unit tests for each of the functions, including edge cases and run-time errors, to ensure the correctness. Due to time constraints we did not yet write tests for the comprehension.

4 Efficiency

From a proper implementation we would expect the run time of a Haskell program to be small. Except from the missing list comprehension we believe that our code is efficient.

5 Robustness

Throughout our code we have chosen to deal with type-correct inputs that are illegal/invalid using an informative error message.

6 Maintainability

WE would argue that our code snippets are reasonable shared through parameterized auxiliary definitions by re-do the same functionality multiple times but re-using when needed. Our main code has a proper lay out, that is, we have chosen to let functions that share a common theme be grouped together. Furthermore, every function in our code has been commented on.

7 Assessment

Despite both of us retaking this class, we tried to re-do the assignment from scratch. And this time around we were not as lost as last year. We did, however, sneak a peak on how to implement `exec` and `execute`. We have a much better understanding of monads, though we wouldn't be able to explain it to a five year old, yet.

8 Appendix

8.1 BOA

```
-- 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 $ \_r -> (Right a, [])
  m >>= f = Comp $ \r ->
    case runComp m r of
      (Left re, s') -> (Left re, s')
      (Right a, s') ->
        case runComp (f a) r of
          (Left re, s'') -> (Left re, s' ++ s'')
          (Right b, s'') -> (Right b, s' ++ 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 ------

{-
abort re is used for signaling the runtime error re
-}
abort :: RunError -> Comp a
abort re = Comp $ \_r -> (Left re, [])

{-
look x returns the current binding of the variable x
(or signals an EBadVar x error if x is unbound)
-}
look :: VName -> Comp Value
look x = Comp $ \r ->
  case lookup x r of
    Nothing -> runComp (abort (EBadVar x)) r
```

```
52      Just value -> (Right value, [])
53
54  {-
55  withBinding x v m runs the computation m with x
56  bound to v, in addition to any other current bindings
57  -}
58  withBinding :: VName -> Value -> Comp a -> Comp a
59  withBinding x v m = Comp $ \r -> runComp m ((x, v):r)
60
61  {-
62  output s appends the line s to the output list.
63  s should not include a trailing newline character
64  (unless the line arises from printing a string value
65  that itself contains an embedded newline)
66  -}
67  output :: String -> Comp ()
68  output s = Comp $ \_r -> (Right (), [s])
69
70
71
72  ------ Helper functions for interpreter ------
73
74  {-
75  truthy v simply determines whether the value v
76  represents truth or falsehood, as previously specified
77  -}
78  truthy :: Value -> Bool
79  truthy NoneVal = False
80  truthy FalseVal = False
81  truthy (IntVal 0) = False
82  truthy (StringVal "") = False
83  truthy (ListVal []) = False
84  truthy _ = True
85
86
87  {-
88  operate o v1 v2 applies the operator o to the arguments
89  v1 and v2, returning either the resulting value, or an error
90  message if one or both arguments are inappropriate for the
91  operation
92
93  operate has changed visual from last year; only In was a true
94  copy-paste from last year w.r.g. to the element search...
95  much simpler than what I was about to venture into -.-
96  -}
97  operate :: Op -> Value -> Value -> Either String Value
98  operate Plus (IntVal e1) (IntVal e2) = Right (IntVal (e1 + e2))
99  operate Minus (IntVal e1) (IntVal e2) = Right (IntVal (e1 - e2))
100 operate Times (IntVal e1) (IntVal e2) = Right (IntVal (e1 * e2))
101 operate Div (IntVal e1) (IntVal e2) = if e2 == 0 then Left "
        attempted division by zero" else Right (IntVal (e1 `div` e2))
102 operate Mod (IntVal e1) (IntVal e2) = if e2 == 0 then Left "
        attempted division by zero" else Right (IntVal (e1 `mod` e2))
103 operate Eq e1 e2 = if e1 == e2 then Right TrueVal else Right
        FalseVal
104 operate Less (IntVal e1) (IntVal e2) = if e1 < e2 then Right
        TrueVal else Right FalseVal
```

```
operate Greater (IntVal e1) (IntVal e2) = if e1 > e2 then Right TrueVal else Right FalseVal
-- has to compare with Eq, hence recursively comparing each element
operate In _ (ListVal []) = Right FalseVal
operate In e1 (ListVal (x:xs)) = if operate Eq e1 x == Right TrueVal then Right TrueVal else operate In e1 (ListVal xs)
operate _ _ _ = Left "invalid value type for operation"


{-
apply f [v1,...,vn] applies the built-in function f to the
(already evaluated) argument tuple v1, ..., vn, possibly
signaling an error if f is not a valid function name (EBadFun),
or if the arguments are not valid for the function (EBadArg)
-}
apply :: FName -> [Value] -> Comp Value
apply "range" vs = compRange vs
apply "print" vs = compPrint vs
apply f _ = abort (EBadFun f)

------ Main functions of interpreter ------

{-
eval e is the computation that evaluates the expression e
in the current environment and returns its value
-}
eval :: Exp -> Comp Value
eval (Const v) = return v
eval (Var x) = look x
eval (Oper o e1 e2) =
  do
  res1 <- eval e1
  res2 <- eval e2
  case operate o res1 res2 of
    Left s -> abort (EBadArg s)
    Right res3 -> return res3
eval (Not e) =
  do
  a <- eval e
  if truthy a then return FalseVal else return TrueVal
eval (Call f en) =
  do
  res <- mapM eval en
  apply f res
eval (List en) =
  do
  res <- mapM eval en
  return (ListVal res)


-- WHAT WE TRIED TO DO FOR COMPR

-- eval (Compr _ [CCFor x e]) =
--   case eval e of
--     (ListVal res) -> withBinding x res
--     _ -> abort (EBadArg "what")
-- eval (Compr _ [CCIf e]) =
```

```
--   if truthy (eval e)
--     then abort (EBadArg "failed guard")
--     else
-- eval (Compr e0 (cc:ccs)) =
--   case cc of
--     (CCFor x e) ->
--       do
--       res <- eval e
--       case res of
--         (ListVal res) -> withBinding x (ListVal res) (eval (
    Compr e0 ccs))
--         _ -> abort (EBadArg "what")
--     (CCIf e) ->
--       do
--       res <- eval e
--       if truthy res
--         then abort (EBadArg "failed guard")
--         else (eval (Compr e0 ccs))


{-
exec p is the computation arising from executing the program
(or program fragment) p, with no nominal return value, but
with any side effects in p still taking place in the computation

SDef x e evaluates e to a value, and binds x to that value for
the remaining statements in the sequence

SExp e just evaluates e and discards the result value
-}
exec :: Program -> Comp ()
exec [] = return ()
exec (sm:sms) =
  case sm of
    (SDef x e) ->
      do
      res <- eval e
      withBinding x res (exec sms)
    (SExp e) ->
      do
      _res <- eval e
      exec sms

{-
execute p explicitly returns the list of output lines, and the
error message (if relevant) resulting from executing p in the
initial environment, which contains no variable bindings
-}
execute :: Program -> ([String], Maybe RunError)
execute p =
  let (u,v) = runComp (exec p) [] in
    case u of
      Right _ -> (v, Nothing)
      Left err -> (v, Just err)


```

```
------ The two built-in functions of BOA + auxillary functions
    ------

-- aux functions for apply
{-
compRange computes the range as foretold by the assignment text
-}
compRange :: [Value] -> Comp Value
compRange vs =
  case vs of
    [IntVal start, IntVal end, IntVal stepSize] ->
      if stepSize == 0
        then abort (EBadArg "step size cannot be zero in range
    function")
        else if start < end && stepSize > 0
          then return (ListVal (map IntVal [start,(start+stepSize)
    ..(end-1)]))
          else if start > end && stepSize < 0
            then return (ListVal (map IntVal [start,(start+stepSize
    )..(end+1)]))
            else return (ListVal [])
    [IntVal start, IntVal end] ->
      if start < end
        then return (ListVal (map IntVal [start..end-1]))
        else return (ListVal (map IntVal [start..end+1]))
    [IntVal end] ->
      return (ListVal (map IntVal [0..(end-1)]))
    _ ->
      abort (EBadArg "incorrect list size or value types for range
    function")


{-
compPrint computes the range as foretold by the assignment text
uses following functions to function properly
 - convertString
 - getSpace
 - convertList
 - convertListElement
 - getComma

-}
compPrint :: [Value] -> Comp Value
compPrint vs =
  do
  output (convertString vs)
  return NoneVal

{-
takes the values given by compPrint and prints the value;
main print function calls on getSpace to, well, get space
and convertList
-}
convertString :: [Value] -> String
convertString [] = ""
convertString (v:vs)=
  case v of
```

```
268     NoneVal -> "None" ++ getSpace (v:vs) ++ convertString vs
269     TrueVal -> "True" ++ getSpace (v:vs) ++ convertString vs
270     FalseVal -> "False" ++ getSpace (v:vs) ++ convertString vs
271     IntVal x -> show x ++ getSpace (v:vs) ++ convertString vs
272     StringVal s -> s ++ getSpace (v:vs) ++ convertString vs
273     ListVal ls -> convertList ls ++ getSpace (v:vs) ++
        convertString vs
274
275 {-
276 returns either a space or none based on the length of the
277 given value
278 -}
279 getSpace :: [Value] -> String
280 getSpace vs = if length vs == 1 then "" else " "
281
282 {-
283 returns "[" "]" when evoked
284 -}
285 convertList :: [Value] -> String
286 convertList vs = "[" ++ convertListElement vs ++ "]"
287
288
289 -- same as convertString but with getComma
290 convertListElement :: [Value] -> String
291 convertListElement [] = ""
292 convertListElement (v:vs) =
293   case v of
294     NoneVal -> "None" ++ getComma (v:vs) ++ convertListElement vs
295     TrueVal -> "True" ++ getComma (v:vs) ++ convertListElement vs
296     FalseVal -> "False" ++ getComma (v:vs) ++ convertListElement vs
297     IntVal x -> show x ++ getComma (v:vs) ++ convertListElement vs
298     StringVal s -> s ++ getComma (v:vs) ++ convertListElement vs
299     ListVal ls -> convertList ls ++ getComma (v:vs) ++
        convertListElement vs
300
301 {-
302 same semantic as getSpace here it returns a ","
303  -}
304 getComma :: [Value] -> String
305 getComma vs = if length vs == 1 then "" else ", "
```

Listing 1: BoaInterp.hs

8.2 Test of BOA

```
-- 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)
    interpreterTests

interpreterTests :: TestTree
interpreterTests = testGroup "Tests for the boa interpreter"
    [testCase "return" $
        runComp (return ()) []
          @?= (Right (),[]),

    -- testCase "abort" $
    --   runComp (do
    --     abort (EBadArg "this is a test")) []
    --     @?= (Left (EBadArg "this is a test"),[]),

    -- ABOVE TEST got ambiguous a0 error, abort gets tested in
    other test cases

    -- monad operation tests
    testCase "look an existing value" $
      runComp (look "x") [("x",(IntVal 5))]
        @?= (Right (IntVal 5),[]),
    testCase "look an non-existing value" $
      runComp (look "x") []
        @?= (Left (EBadVar "x"),[]),
    testCase "look after withBinding" $
      runComp (withBinding "x" (IntVal 5) (look "x")) []
        @?= (Right (IntVal 5),[]),
    testCase "output" $
      runComp (output "test") []
        @?= (Right (),["test"]),
    testCase "output two strings" $
      runComp (do
        _ <- output "test"
        _ <- output "test"
        return ()) []
        @?= (Right (),["test", "test"]),

    -- truthy tests
    testCase "truthy none" $
      truthy NoneVal
        @?= False,
    testCase "truthy false" $
      truthy FalseVal
        @?= False,
    testCase "truthy 0" $
```

```
      truthy (IntVal 0)
        @?= False,
    testCase "truthy empty string" $
      truthy (StringVal "")
        @?= False,
    testCase "truthy empty list" $
      truthy (ListVal [])
        @?= False,
    testCase "truthy true" $
      truthy (TrueVal)
        @?= True,
    testCase "truthy 1" $
      truthy (IntVal 1)
        @?= True,
    testCase "truthy string" $
      truthy (StringVal "yes")
        @?= True,
    testCase "truthy list" $
      truthy (ListVal [IntVal 1])
        @?= True,

    -- operate tests
    testCase "operate plus" $
      operate Plus (IntVal 1) (IntVal 1)
        @?= Right (IntVal 2),
    testCase "operate plus wrong value type" $
      operate Plus (IntVal 5) (NoneVal)
        @?= Left "invalid value type for operation",
    testCase "operate minus" $
      operate Minus (IntVal 1) (IntVal 1)
        @?= Right (IntVal 0),
    testCase "operate minus wrong value type" $
      operate Minus (IntVal 5) (NoneVal)
        @?= Left "invalid value type for operation",
    testCase "operate times" $
      operate Times (IntVal 3) (IntVal 1)
        @?= Right (IntVal 3),
    testCase "operate times wrong value type" $
      operate Times (IntVal 5) (NoneVal)
        @?= Left "invalid value type for operation",
    testCase "operate div" $
      operate Div (IntVal 6) (IntVal 2)
        @?= Right (IntVal 3),
    testCase "operate div zero" $
      operate Div (IntVal 1) (IntVal 0)
        @?= Left "attempted division by zero",
    testCase "operate div wrong value type" $
      operate Div (IntVal 5) (NoneVal)
        @?= Left "invalid value type for operation",
    testCase "operate mod" $
      operate Mod (IntVal 7) (IntVal 2)
        @?= Right (IntVal 1),
    testCase "operate mod zero" $
      operate Mod (IntVal 1) (IntVal 0)
        @?= Left "attempted division by zero",
    testCase "operate mod wrong value type" $
      operate Mod (IntVal 5) (NoneVal)
```

```
      @?= Left "invalid value type for operation",
  testCase "operate eq true" $
    operate Eq (IntVal 6) (IntVal 6)
      @?= Right TrueVal,
  testCase "operate eq false" $
    operate Eq (TrueVal) (FalseVal)
      @?= Right FalseVal,
  testCase "operate less true" $
    operate Less (IntVal 6) (IntVal 8)
      @?= Right TrueVal,
  testCase "operate less false" $
    operate Less (IntVal 6) (IntVal 5)
      @?= Right FalseVal,
  testCase "operate less wrong value type" $
    operate Less (IntVal 5) (NoneVal)
      @?= Left "invalid value type for operation",
  testCase "operate greater true" $
    operate Greater (IntVal 6) (IntVal 5)
      @?= Right TrueVal,
  testCase "operate greater false" $
    operate Greater (IntVal 6) (IntVal 8)
      @?= Right FalseVal,
  testCase "operate greater wrong value type" $
    operate Greater (IntVal 5) (NoneVal)
      @?= Left "invalid value type for operation",
  testCase "operate in true" $
    operate In (IntVal 5) (ListVal [(IntVal 5)])
      @?= Right TrueVal,
  testCase "operate in false" $
    operate In (IntVal 5) (ListVal [(IntVal 8)])
      @?= Right FalseVal,
  testCase "operate in empty list" $
    operate In (IntVal 5) (ListVal [])
      @?= Right FalseVal,
  testCase "operate in wrong value type" $
    operate In (IntVal 5) (NoneVal)
      @?= Left "invalid value type for operation",

  -- range tests
  testCase "range, 3 args, n3 > 0" $
    runComp (apply "range" [IntVal 1,IntVal 5,IntVal 2]) []
      @?= (Right (ListVal [IntVal 1,IntVal 3]),[]),
  testCase "range, 3 args, n3 > 0, n1 > n2" $
    runComp (apply "range" [IntVal 2,IntVal 1,IntVal 2]) []
      @?= (Right (ListVal []),[]),
  testCase "range, 3 args, n3 < 0" $
    runComp (apply "range" [IntVal 5,IntVal 1,IntVal (-2)]) []
      @?= (Right (ListVal [IntVal 5,IntVal 3]),[]),
  testCase "range, 3 args, n3 < 0, n1 < n2" $
    runComp (apply "range" [IntVal 1,IntVal 2,IntVal (-2)]) []
      @?= (Right (ListVal []),[]),
  testCase "range, 3 args, n3 == 0" $
    runComp (apply "range" [IntVal 1,IntVal 5,IntVal 0]) []
      @?= (Left (EBadArg "step size cannot be zero in range
function"),[]),
  testCase "range, 3 args, incorrect value type" $
    runComp (apply "range" [IntVal 1,TrueVal,IntVal 1]) []
```

```
167         @?= (Left (EBadArg "incorrect list size or value types for
    range function"),[]),
168     testCase "range, 2 args, n1 < n2" $
169       runComp (apply "range" [IntVal 1,IntVal 3]) []
170         @?= (Right (ListVal [IntVal 1,IntVal 2]),[]),
171     testCase "range, 2 args, n1 > n2" $
172       runComp (apply "range" [IntVal 3,IntVal 1]) []
173         @?= (Right (ListVal []),[]),
174     testCase "range, 2 args, incorrect value type" $
175       runComp (apply "range" [IntVal 1,TrueVal]) []
176         @?= (Left (EBadArg "incorrect list size or value types for
    range function"),[]),
177     testCase "range, 1 arg" $
178       runComp (apply "range" [IntVal 3]) []
179         @?= (Right (ListVal [IntVal 0,IntVal 1,IntVal 2]),[]),
180     testCase "range, 1 arg, incorrect value type" $
181       runComp (apply "range" [TrueVal]) []
182         @?= (Left (EBadArg "incorrect list size or value types for
    range function"),[]),
183
184     -- print tests
185     testCase "print simple values" $
186       runComp (apply "print" [NoneVal, TrueVal, FalseVal, IntVal
    0]) []
187         @?= (Right NoneVal,["None True False 0"]),
188     testCase "print two strings" $
189       runComp (apply "print" [StringVal "string 1", StringVal "
    string 2"]) []
190         @?= (Right NoneVal,["string 1 string 2"]),
191     testCase "print listval" $
192       runComp (apply "print" [ListVal [StringVal "string", IntVal
    0]]) []
193         @?= (Right NoneVal,["[string, 0]"]),
194     testCase "print nested listval" $
195       runComp (apply "print" [ListVal [ListVal [IntVal 1], ListVal
    [IntVal 2]]]) []
196         @?= (Right NoneVal,["[[1], [2]]"]),
197     testCase "print empty list" $
198       runComp (apply "print" []) []
199         @?= (Right NoneVal,[""]),
200     testCase "print complex" $
201       runComp (apply "print" [IntVal 42, StringVal "foo", ListVal [
    TrueVal, ListVal []], IntVal (-1)]) []
202         @?= (Right NoneVal,["42 foo [True, []] -1"]),
203
204     -- eval tests
205     testCase "eval with const" $
206       runComp (eval (Const (IntVal 5))) []
207         @?= (Right (IntVal 5),[]),
208     testCase "eval with var" $
209       runComp (eval (Var "x")) [("x",(IntVal 5))]
210         @?= (Right (IntVal 5),[]),
211     testCase "eval with oper" $ -- operate helper function already
    extensively tested
212       runComp (eval (Oper Plus (Const (IntVal 1))(Const (IntVal 1))
    )) []
213         @?= (Right (IntVal 2),[]),
```

```
214     testCase "eval with oper runerror" $
215       runComp (eval (Oper Div (Const (IntVal 1))(Var "x"))) [("X",
        NoneVal)]
216         @?= (Left (EBadVar "x"),[]),
217     testCase "eval with oper propagated runerror" $
218       runComp (eval (Oper Div (Const (IntVal 1))(Var "x"))) []
219         @?= (Left (EBadVar "x"),[]),
220     testCase "eval with not (false)" $ -- truthy helper function
        already extensively tested
221       runComp (eval (Not (Const (IntVal 0)))) []
222         @?= (Right TrueVal,[]),
223     testCase "eval with not (true)" $
224       runComp (eval (Not (Const (IntVal 1)))) []
225         @?= (Right FalseVal,[]),
226     testCase "eval with call range" $ -- range function already
        extensively tested
227       runComp (eval (Call "range" [Const (IntVal 1), Const (IntVal
        5), Const (IntVal 2)])) []
228         @?= (Right (ListVal [IntVal 1,IntVal 3]),[]),
229     testCase "eval with call range error" $
230       runComp (eval (Call "range" [Const (IntVal 1), Const (IntVal
        5), Const NoneVal])) []
231         @?= (Left (EBadArg "incorrect list size or value types for
        range function"),[]),
232     testCase "eval with call print" $ -- print function already
        extensively tests
233       runComp (eval (Call "print" [Const (IntVal 42), Const (
        StringVal "foo"), Const (ListVal [TrueVal, ListVal []]), Const
        (IntVal (-1))])) []
234         @?= (Right NoneVal,["42 foo [True, []] -1"]),
235     testCase "eval with call bad function" $
236       runComp (eval (Call "foo" [Const (IntVal 0)])) []
237         @?= (Left (EBadFun "foo"),[]),
238     testCase "eval with list empty" $
239       runComp (eval (List [])) []
240         @?= (Right (ListVal []),[]),
241     testCase "eval with list runerror" $
242       runComp (eval (List [Var "x"])) []
243         @?= (Left (EBadVar "x"),[]),
244     testCase "eval with list runerror left to right" $
245       runComp (eval (List [Var "x",Var "y"])) []
246         @?= (Left (EBadVar "x"),[]),
247     --TODO: compr tests
248
249     -- exec tests
250     testCase "exec sdef" $
251       runComp (exec [SDef "x" (Const (IntVal 1))]) []
252         @?= (Right (),[]),
253     testCase "exec sexp" $
254       runComp (exec [SExp (Const (IntVal 1))]) []
255         @?= (Right (),[]),
256     testCase "exec output" $
257       runComp (exec [SExp (Call "print" [Const (StringVal "Hello")
        ])]) []
258         @?= (Right (),["Hello"]),
259     testCase "exec sdef + sexp + output" $
260       runComp (exec [SDef "x" (Const (StringVal "Hello")), SExp (
```

```
    Call "print" [Var "x"])]) []
261       @?= (Right (),["Hello"]),
262   testCase "exec runerror" $
263     runComp (exec [SExp (Var "x")]) []
264       @?= (Left (EBadVar "x"),[]),
265
266   -- execute tests
267   testCase "execute sdef" $
268     execute [SDef "x" (Const (IntVal 1))]
269       @?= ([], Nothing),
270   testCase "execute sexp" $
271     execute [SExp (Const (IntVal 1))]
272       @?= ([], Nothing),
273   testCase "execute output" $
274     execute [SExp (Call "print" [Const (StringVal "Hello")])]
275       @?= (["Hello"], Nothing),
276   testCase "execute sdef + sexp + output" $
277     execute [SDef "x" (Const (StringVal "Hello")), SExp (Call "
    print" [Var "x"])]
278       @?= (["Hello"], Nothing),
279   testCase "execute runerror" $
280     execute [SExp (Var "x")]
281       @?= ([], Just (EBadVar "x")),
282
283   -- example ast tests
284   testCase "execute misc.ast from handout" $
285     do
286       pgm <- read <$> readFile "examples/misc.ast"
287       out <- readFile "examples/misc.out"
288       execute pgm @?= (lines out, Nothing),
289   testCase "execute crash.ast from handout" $
290     do
291       pgm <- read <$> readFile "examples/crash.ast"
292       out <- readFile "examples/crash.out"
293       execute pgm @?= (lines out, Nothing)]
```

Listing 2: Test.hs