Advanced Programming
Assignment 2 : A Boa Interpreter
Report

AP22 Assignment 2 Group 60
KRW521, CPJ395

September 22, 2022

1 Design and Implementation

1.1 Monad operations

First we constructed the Comp monad and its executable runComp. **runComp** is a function that accepts a Comp monad and an environment shaped like [**(VName, Value)**] , and returns a tuple **(Either RunError a, [String])**. Comp monad's return puts the received value at the head of the tuple and creates the empty list [ ] at the end of the tuple. (>>=) applies the operating function to be implemented next to the old Comp monad, inherits the error, or performs the operation. And appends the corresponding output to the [String] at the end of the tuple. It is implemented as follows:

```
instance Monad Comp where
    return a = Comp (\_ -> (Right a, mempty))
    x >>= f = Comp (\env -> case runComp x env of
            (Left err, s1) -> (Left err, s1)
            (Right x1, s1) -> case runComp (f x1) env of
                    (Left err, s2) -> (Left err, s1 `mappend` s2)
                    (Right x2, s2) -> (Right x2, s1 `mappend` s2))
```

We then implement the corresponding operator functions, which include the **about** function to create an error, the **look** function to get the value of a variable from the environment, the **withBinding** function to assign a value to a variable, and the **output** function to output a string. When we build the look function we use a **getValue** helper function. The getValue function will take a variable name and an environment and search for the value bound to that variable in the environment from scratch. If the search returns **Just Value** if it fails it returns **Noting**. Note that here we tightly let the function search from the beginning and do not exclude the case where there may be two identical variable bindings in an environment. This is because, when we bind variables using withBinding, the function always puts the latest binding at the head of the environment list, as shown below. So we can simply facilitate the environment list from the beginning and find the latest binding.

```
    withBinding :: VName -> Value -> Comp a -> Comp a
    withBinding vn v m = Comp (\env -> runComp m ((vn, v):env))
```

1.2 Auxilary functions for interpreter

The main purpose of the helper function is to handle several problems that are not well handled in the eval function.

The **truthy** function converts specific values to the Boolean value False in order to facilitate subsequent operations on them.

The **operate** functions are used to handle mathematical operations, including **Plus**, **Minus**, **Times**, **Div**, **Mod**, **Eq**, **Less**, **Greater**, and **In**. In fact, we can do most of these functions by simply binding them to Haskell's basic library. However, it should be noted that the second term of the two division-related operations (**Div** and **Mod**) cannot be **zero**, and the **Eq** and **In** operations can be performed on non-integer types.

The **apply** function takes a string command (**"range"** or **"print"**) and an associated parameter [**Value**], and then passes back **Comp Value**. **"range"** allows the user to automatically complete the corresponding statement on default input, which can be achieved using pattern recognition. To implement the range function, we construct the helper function **rangeFunc**. The rangFunc function takes three arguments and returns the result we need, which is implemented recursively. Note that the **rangeFunc** function can only output values of the form [**Value**], but we need the output of the apply function to be of the type **Comp Value**. So we need to use the **lambda function** to convert the output of rangeFunc to the **Value** type we need and finally put it into **Comp monad**. The specific approach is shown below.

```
Comp (\_ -> (Right (ListVal (rangeFunc e0 e1 e2)), mempty))
```

"print" wants to print out the [**Value**] parameter. To do this, we construct the **printFunc** helper function. printFunc uses **(x:xs)** to match the elements at the head of the list and appends them one by one to the **String** recursively, and finally outputs the String. It is important to note that **Value** contains a **ListVal** [**Value**] type, which means that we may take out a **new** [**Value**] when fetching elements. However, according to the requirements of the question, the [Value] in [Value] is not printed in the same format as itself, so we use a simple recursion to handle this case. Our solution is to construct a **printValue** function much like **printFunc**, they differ only in the addition of separators, printFunc uses **" "** to separate elements, while printValue uses **", "** to separate elements.

1.3 Main functions of interpreter

The **eval** function calculates the input **Exp**. It gets the different keywords by pattern matching and performs the corresponding processing.

For **Const x eval** function returns it directly as **Comp x**. For variable **Var x** the **eval** function will call the **look** function and return its bound value, or return an error if it is not bound, as handled in **look**.

For **Oper**, since its two arguments are also of type **Exp**, the program will recursively call **eval** to compute the values of the two expressions. At the end of the calculation, the actual values of the two expressions are retrieved using the **do** block, and the auxiliary function **operate** is called to calculate the result. If the return value is **Right m**, then **return m**, and if the return value is **Left m**, then **operator** has an error, and the program will call the **about** function to throw the corresponding error.

For **Not**, since Not's argument is also an expression of type **Exp**, the program recursively calls the **eval** function to compute this expression. Then we call the helper function **truthy** to filter out the error value. Then the error value we defined is returned as **FalseVal** and the correct value is returned as **TrueVal** by the **case** statement.

For **Cell**, since Cell has one of the parameters [**Exp**], we also need to recursively call **eval** to calculate each item in [Exp] and finally return it as **ListVal** [**Value**]. The **case** statement matches the **ListVal** [**Value**], takes the argument [**Value**] needed by the apply function and finally calls the **apply** function to get the return value.

As stated above, **eval List** [**Exp**] will compute each element in [**Exp**] and eventually return a **ListVal** [**Value**]. We divide the list into head and tail by **(x:xs)** and process the head part recursively. The recursive threshold condition is set to return an empty List when the List is empty, and eventually, the result is put into this empty List to get the return value [**Value**]. Because **eval** needs the return value to be **Comp Value**, we also need to add **ListVal** before [Value] to make it become **Value**.

To realize the list comprehension, we just using the most intuitive way. For CCFor, we judge the expression if it is a list, if not then throw an error; if is then continue the program by using **ere $<-$ withBinding vn e1 (eval (Compr e cs))** in recursion and then continue the CCFor by using **ere1**

<-eval (Compr e (CCFor vn (Const (ListVal es1)):cs)), and then splice these two together. As for CCIf, we just judge the expression is true or not by using function **truthy**, if is, then continue evaluate the remaining expression; if not, then **return (ListVal [])** to return to the upper layer in [**CClause**].

For **exec** function, we first use the pattern matching to see if the **Program** is an empty list, if is, then **return mempty**, if not, then we will check what is the first **stmt**. For **SDef**, we fist evaluate the expression as x and use **withBinding** to attach the **Value x** to the name, then make it compute in **exec** remaining part by writing **withBinding v x (exec xs)**, which is a kind of recursion. For **SExp**, we don't need to attach the variable name to Value, so we just evaluate the expression by **eval e**, then **exec** the remaining part by recursion.

As for the **execute** function that stands in a higher place, while **execute x**, we just need to run exec in a starting environment, that is, **runComp (exec x)** [ ], and see if there is a **RunError** thrown in executing by observing the result of **runComp (exec x)** [ ]. So we can decide what to be output in the **execute** function.

2 Assessment of The Code

2.1 Completeness

All functions are completed, and the completion of all functions are as follows:

Class of Function	Function Name	Completion
Monad operations	abort	Completed
Monad operations	look	Completed
Monad operations	withBinding	Completed
Monad operations	output	Completed
Auxilary functions	truthy	Completed
Auxilary functions	operate	Completed
Auxilary functions	apply	Completed
Interpreter functions	eval	Completed
Interpreter functions	exec	Completed
Interpreter functions	execute	Completed

2.2 Correctness

All functions have been judged by **OnlineTA**, and the feedback by OnlineTA is all "**OK**" except 3 warnings and 2 errors in the list comprehension in **eval**. After analysis, we find out the problem appears in this line in our code about **eval Compr** using **CCFor** : **ere <- withBinding vn e1 (eval (Compr e cs))**, that is, **in some conditions** the value cannot be fixed to the value name because the program set the priority to executing **(eval (Compr e cs))** that make the program continue to execute in an environment that doesn't contain the new name-value pair. But, we also find out that in some test cases using CCFor in list comprehension, our program do get the right answers. After referring some materials about Haskell, we speculate that our errors and warnings may relate to the compiling mechanism of Haskell (like laziness). And our evaluation is as follows:

Class of Function	Function Name	Test Result
Monad operations	abort	OK
Monad operations	look	OK
Monad operations	withBinding	OK
Monad operations	output	OK
Auxiliary functions	truthy	OK
Auxiliary functions	operate	OK
Auxiliary functions	apply	OK
Interpreter functions	eval	3 warnings, 2 errors, remaining OK
Interpreter functions	exec	OK
Interpreter functions	execute	OK

2.3 Efficiency

All functions run as expected in our tests. For example, the time-consuming of functions such as **Plus, Minus, Times, Div, Mod, Eq, Less, Greater, In** in **operate** is basically the same as the corresponding functions in the Haskell basic function library, and no abnormality was observed under the stress test. The other functions like **abort, look, withBinding, output...** can also be executed and return the result within quite short time. The evaluation of function efficiency is as follows:

Class of Function	Function Name	Run Time
Monad operations	abort	Normal
Monad operations	look	Normal
Monad operations	withBinding	Normal
Monad operations	output	Normal
Auxiliary functions	truthy	Normal
Auxiliary functions	operate	Normal
Auxiliary functions	apply	Normal
Interpreter functions	eval	Normal
Interpreter functions	exec	Normal
Interpreter functions	execute	Normal

2.4 Robustness

As is mentioned in the assignment, while receiving some input that may be Haskell-type-correct (like division by zero and mod by zero), our program will output **Left "error message"** instead of evoking the error in Haskell. And another example in our program is while receiving the error number of input parameter in calling "range" function, our program will also output the error message by using **abort** instead of evoking the error in Haskell. And there are other mechanisms in our program to ensure the robustness. So we do have confidence in the robustness evaluation of our program. The evaluation of function Robustness is as follows:

Class of Function	Function Name	Robustness
Monad operations	abort	Strong
Monad operations	look	Strong
Monad operations	withBinding	Strong
Monad operations	output	Strong
Auxiliary functions	truthy	Strong
Auxiliary functions	operate	Strong
Auxiliary functions	apply	Strong
Interpreter functions	eval	Good
Interpreter functions	exec	Strong
Interpreter functions	execute	Strong

2.5 Maintainability

The key functions in the code are all commented. This time, our code respect the monadic abstraction by only expressing the required functionality through the associated-operation functions (like **abort, look**, etc.) rather than relying directly on the implementation of Comp. Also using monad in our code make the code concise and neat. Overall, our code is readable and easy to maintain. Our evaluation of code maintainability are as follows:

Class of Function	Function Name	Maintainability
Monad operations	abort	Good
Monad operations	look	Good
Monad operations	withBinding	Good
Monad operations	output	Good
Auxiliary functions	truthy	Good
Auxiliary functions	operate	Good
Auxiliary functions	apply	Good
Interpreter functions	eval	Not bad
Interpreter functions	exec	Good
Interpreter functions	execute	Good

A Appendix: BoaInterp.hs

```haskell
-- 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 (\_ -> (Right a, mempty))
  x >>= f = Comp (\env -> case runComp x env of
                          (Left err, s1) -> (Left err, s1)
                          (Right x1, s1) -> case runComp (f x1) env of
                                              (Left err, s2) -> (Left err, s1 `mappend` s2)
                                              (Right x2, s2) -> (Right x2, s1 `mappend` s2))

-- 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 :: RunError -> Comp a
abort err = Comp (\_ -> (Left err, mempty))

look :: VName -> Comp Value
look vn = Comp (\env -> case getValue vn env of
                          Nothing -> (Left (EBadVar vn), mempty)
                          Just x   -> (Right x, mempty))

getValue :: VName -> Env -> Maybe Value
getValue _ [] = Nothing
getValue vn env
  | vn == fst (head env) = Just (snd $ head env)
  | otherwise             = getValue vn (tail env)


withBinding :: VName -> Value -> Comp a -> Comp a
withBinding vn v m = Comp (\env -> runComp m ((vn, v):env))

output :: String -> Comp ()
output s = Comp (\_ -> (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 x) (IntVal y)  = Right (IntVal (x + y))
operate Minus (IntVal x) (IntVal y) = Right (IntVal (x - y))
operate Times (IntVal x) (IntVal y) = Right (IntVal (x * y))
operate Div (IntVal x) (IntVal y)
  | IntVal y == IntVal 0            = Left "Div Zero"
  | otherwise                       = Right (IntVal (x `div` y))
operate Mod (IntVal x) (IntVal y)
  | IntVal y == IntVal 0            = Left "Mod Zero"
  | otherwise                       = Right (IntVal (x `mod` y))
```

```haskell
operate Eq x y
  | x == y                    = Right TrueVal
  | otherwise                 = Right FalseVal
operate Less (IntVal x) (IntVal y)
  | x < y                     = Right TrueVal
  | otherwise                 = Right FalseVal
operate Greater (IntVal x) (IntVal y)
  | x > y                     = Right TrueVal
  | otherwise                 = Right FalseVal
operate In x (ListVal y)
  | x `elem` y                = Right TrueVal
  | otherwise                 = Right FalseVal
operate _ _ _                 = Left "Operate Error"

apply :: FName -> [Value] -> Comp Value
-- "range"
apply "range" [IntVal e0]                       = apply "range" [IntVal 0, IntVal e0, IntVal 1]
apply "range" [IntVal e0, IntVal e1]            = apply "range" [IntVal e0, IntVal e1, IntVal 1]
apply "range" [_, _, IntVal 0]   = abort (EBadArg "Illegal Input: zero")
apply "range" [IntVal e0, IntVal e1, IntVal e2] = Comp (\_ -> (Right (ListVal (rangeFunc (IntVal e0) (IntVal e1) (IntVal e2))), mempty))
apply "range" _                                 = abort (EBadArg "Syntax Error: range")
-- "print"
apply "print" s =
  do
  {
    output (printFunc s);
    return NoneVal
  }
-- Err
apply x _ = abort (EBadFun x)

rangeFunc :: Value -> Value -> Value -> [Value]
rangeFunc (IntVal e0) (IntVal e1) (IntVal e2)
  | e0 >= e1 && e2 > 0 = []
  | e0 <= e1 && e2 < 0 = []
```

```
  | otherwise          = IntVal e0 : rangeFunc (IntVal (e0 + e2)) (IntVal
      e1) (IntVal e2)
rangeFunc _ _ _        = error "Not Int"

printFunc :: [Value] -> String
printFunc [] = ""
printFunc [x]            = printValue x
printFunc (x:xs)         = printValue x ++ " " ++ printFunc xs

printValue :: Value -> String
printValue NoneVal       = "None"
printValue TrueVal       = "True"
printValue FalseVal      = "False"
printValue (IntVal x)    = show x
printValue (StringVal x) = x
printValue (ListVal x)   = "[" ++ getListValue x ++ "]"

getListValue :: [Value] -> String
getListValue [] = ""
getListValue [x]         = printValue x
getListValue (x:xs)      = printValue x ++ ", " ++ getListValue xs

-- Main functions of interpreter
eval :: Exp -> Comp Value
eval (Const x) = return x
eval (Var x)   = look x
eval (Oper op e0 e1) =
  do
    {
      x <- eval e0;
      y <- eval e1;
      case operate op x y of
        (Right m) -> return m
        (Left m)  -> abort (EBadArg m)
    }
eval (Not e) =
  do
    {
      x <- eval e;
      if truthy x
        then return FalseVal
```

```
      else
        return TrueVal
    }
eval (Call f e) =
  do
    {
      x <- eval (List e);
      case x of
        (ListVal y) -> apply f y
        _           -> error "Call Error"
    }
eval (List []) = return (ListVal [])
eval (List (x:xs)) =
  do
    {
      y  <- eval x;
      ys <- eval (List xs);
      case ys of
        (ListVal zs) -> return (ListVal (y:zs))
        _            -> error "List Error"
    }
eval (Compr e []) =
  do
    re<-eval e
    return (ListVal [re])

eval (Compr e ((CCFor vn exp):cs)) =
  do{
    ex<-eval exp;
    let ex1= ex in
      if isListVal ex1
        then
          if cs /= []
            then
              let (e1:es1)=extractList ex1 in
                do
                  ere <- withBinding vn e1 (eval (Compr e cs))
                  ere1 <-eval (Compr e (CCFor vn (Const (ListVal es1)):
                      cs))
                  return (ListVal (extractList ere++extractList ere1))
            else
```

```
                  do
                    re<-eval e
                    return (ListVal [re])
            else
              return (ListVal [])

  }
eval (Compr e ((CCIf exp):cs)) =
  do{
    ex<-eval exp;
    if cs /= []
      then
        if truthy ex
          then
            eval (Compr e cs)
          else
            return (ListVal [])
      else
        if truthy ex
          then
            do
              re<-eval e
              return (ListVal [re])
          else
            return (ListVal [])


  }
isListVal :: Value -> Bool
isListVal (ListVal (x:xs)) = True
isListVal _ = False

extractList :: Value -> [Value]
extractList (ListVal x) = x
extractList x = []

exec :: Program -> Comp ()
exec [] = return mempty
exec ((SDef v e):xs) =
  do
    {
```

```
      x <- eval e;
      withBinding v x (exec xs)
    }
exec ((SExp e):xs) =
  do
    {
      eval e;
      exec xs
    }

execute :: Program -> ([String], Maybe RunError)
execute x =
  case fst (runComp (exec x) []) of
    (Right _) -> (snd (runComp (exec x) []), Nothing)
    (Left y)  -> (snd (runComp (exec x) []), Just y)
```

B Appendix: Test.hs

```
-- 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 "Stubby tests"
  [
    testCase "crash test" $
    execute [SExp (Call "print" [Oper Plus (Const (IntVal 2))
                                           (Const (IntVal 2))]),
             SExp (Var "hello")]
      @?= (["4"], Just (EBadVar "hello")),

```

```
testCase "execute misc.ast from handout" $
 do pgm <- read <$> readFile "examples/misc.ast"
    out <- readFile "examples/misc.out"
    execute pgm @?= (lines out, Nothing),

testCase "test1" $
runComp (look "x")
    [("x", IntVal 3),("y", IntVal 4)]
  @?= (Right (IntVal 3),[]),

testCase "test2" $
runComp (look "y")
    [("x", IntVal 3),("y", IntVal 4)]
  @?= (Right (IntVal 4),[]),

testCase "test3" $
runComp (look "z")
    [("x", IntVal 3),("y", IntVal 4)]
  @?= (Left (EBadVar "z"),[]),

testCase "test4" $
runComp (output "Hello, world")
    [("x", IntVal 3),("y", IntVal 4)]
  @?= (Right (),["Hello, world"]),

testCase "test5" $
runComp (withBinding "z" (IntVal 3)
    (look "z")) [("x", IntVal 3),("y", IntVal 4)]
  @?= (Right (IntVal 3),[]),

testCase "test6" $
runComp (abort (EBadVar "Bad Var"))
    [("x", IntVal 3),("y", IntVal 4)]
  @?= (Left (EBadVar "Bad Var"),[]),

testCase "test7" $
truthy NoneVal @?= False,

testCase "test8" $
(IntVal (-1)) @?= True,
```

```
testCase "test9" $
truthy (ListVal []) @?= False,

testCase "test10" $
truthy (ListVal [ListVal []]) @?= True,

testCase "test11" $
operate Plus (IntVal 5)
    (IntVal 6) @?= Right (IntVal 11),

testCase "test12" $
operate Minus (IntVal 5) (IntVal 6)
  @?= Right (IntVal (-1)),

testCase "test13" $
operate Times (IntVal 5) (IntVal 6)
  @?= Right (IntVal 30),

testCase "test14" $
operate Div (IntVal 5) (IntVal 0)
  @?= Left "Div Zero",

testCase "test15" $
operate Mod (IntVal 5) (IntVal 0)
  @?= Left "Mod Zero",

testCase "test16" $
operate Eq (IntVal 5) (IntVal 6)
  @?= Right FalseVal,

testCase "test17" $
operate Less (IntVal 5) (IntVal 6)
  @?= Right TrueVal,

testCase "test18" $
operate Greater (IntVal 5) (IntVal 6)
  @?= Right FalseVal,

testCase "test19" $
operate In (IntVal 5)
    (ListVal [IntVal 5, IntVal 6])
```

```
      @?= Right TrueVal,

    testCase "test20" $
    operate In (IntVal 5)
        (ListVal [IntVal 9, IntVal 6])
      @?= Right FalseVal,

    testCase "test21" $
    runComp (apply "range" [IntVal 5, IntVal 6])
        [("x", IntVal 4),("y", IntVal 5)]
      @?= (Right (ListVal [IntVal 5]),[]),

    testCase "test22" $
    runComp (apply "range" [IntVal 5])
        [("x", IntVal 4),("y", IntVal 5)]
      @?= (Right
  (ListVal [IntVal 0,IntVal 1,IntVal 2,IntVal 3,IntVal 4]),[])  ,

    testCase "test23" $
    runComp (apply "range"
        [IntVal 5, IntVal 9, IntVal 1])
        [("x", IntVal 4),("y", IntVal 5)]
      @?= (Right
  (ListVal [IntVal 5,IntVal 6,IntVal 7,IntVal 8]),[]) ,

    testCase "test24" $
    runComp (apply "range"
        [IntVal 5, IntVal 9, NoneVal])
        [("x", IntVal 4),("y", IntVal 5)]
      @?= ((Left
  (EBadArg "Syntax Error: range"),[]),

    testCase "test25" $
    runComp (apply "print"
        [IntVal 5, IntVal 9, NoneVal])
        [("x", IntVal 4),("y", IntVal 5)]
      @?= (Right NoneVal,["5 9 None"]),

    testCase "test26" $
    runComp (eval (Const (StringVal "str")))
        [("x", IntVal 3),("y", IntVal 4)]
```

```
      @?= (Right (StringVal "str"),[]),

    testCase "test27" $
    runComp (eval (Var "x"))
        [("x", IntVal 3),("y", IntVal 4)]
      @?= (Right (IntVal 3),[]),

    testCase "test28" $
    runComp (eval (Var "z"))
        [("x", IntVal 3),("y", IntVal 4)]
      @?= (Left (EBadVar "z"),[]),

    testCase "test29" $
    runComp (eval
        (Oper Times (Var "x") (Const (IntVal 6))))
        [("x", IntVal 3),("y", IntVal 4)]
      @?= (Right (IntVal 18),[]),

    testCase "test30" $
    runComp (eval (Call "range"
        [Const (IntVal 2)])) [("x", IntVal 3),("y", IntVal 4)]
      @?= (Right
  (ListVal [IntVal 0,IntVal 1]) ,[]) ,

    testCase "test31" $
    runComp (eval (Call "print"
        [Const (IntVal 4), Const (IntVal (-4)), Const (IntVal (-1))]))
        [("x", IntVal 3),("y", IntVal 4)]
      @?= (Right NoneVal,["4 -4 -1"]),

    testCase "test32" $
    runComp (eval (Compr
        (List [Var "z", Var "y"])   [ (CCFor "z" (Const
        (ListVal [IntVal 5] ))), (CCFor "y" ( Const
        (ListVal [IntVal 5] )))]))
        [("x", IntVal 4),("y", IntVal 5)]
      @?= (Right (ListVal
   [ListVal [IntVal 5,IntVal 5]]) ,[])  ),

    testCase "test33" $
    runComp (exec [SExp (Call "range"
```

```
      [Const (IntVal 2)]) ])
      [("x", IntVal 4),("y", IntVal 5)]
    @?= (Right (),[]),

  testCase "test34" $
  runComp (exec
      [SDef "x" (Call "range" [Const (IntVal 2)]) ])
      [("x", IntVal 4),("y", IntVal 5)]
    @?= (Right (),[]),

  testCase "test35" $
  execute [SExp (Call "range"
      [Const (IntVal 2)]) ]
    @?= ([],Nothing),

  testCase "test36" $
  execute [SDef "x"
      (Call "range" [Const (IntVal 2)])]
    @?= ([],Nothing)]
```