Assignment 1 - Boa Interpreter

Kalle Kromann & Bjørn Christian Vedel Bennetsen

September 2019

1 Design and implementation choices

Comp monad The Comp monad is used to give computations access to the environment and to be able to give an output before the program is finished running. The value in the monad can either be an error (Left) or a value of the expected type (Right). `return a` returns a monad where the value is 'Right a' and with an empty output list, since `return` does not have any output. The bind operator $m >>= f$ unwraps the value a of m and the list of outputs associated with this value, checks whether the value is an error in which case the error is just carried on to the new monad after the bind. If it is not an error it applies the function f to the value by running the computation with the enviroment argument, and uses the result of this computation for the new monad value. The output of the computation is then concatenated with the output associated with the original value.
The rest of the monad operators are meant to provide functions for working with the Comp monad, so we can avoid using statements like `Comp(λ_ -> (Right "something", []))` directly in the `eval` function.
The `abort` function takes a `RunError` and wraps it in a Comp and returns it. The `look`-function takes a variable-name and recursively searches through the environment to see if the variable name is bound to some value. If the variable name is bound it returns a new Comp with the Value wrapped in it, otherwise it returns the `RunError`, `EBadVar`.
The `withBinding` function takes a variable name, a value and a computation. It then create a new environment where it binds the value to the variable name and then runs the computation given in the new environment. The `output` function takes a string and returns a Comp with the output of the computation wrapped in it, providing a way to just put text directly in the Comp monad.

Helper functions The `truthy` helper function evaluates a `Value`-type to a bool. The assignment description asked to let empty strings, empty list, the number 0 and `NoneVal` represent falsehood. So evaluating `truthy (ListVal [])` would return false. The only place we use this helper function is in the list-comprehension evaluation.
The `operate` helper function is intended for use in the `eval` function to evaluate an operation between two values and wrap the result in a new `Value` depending on the operator. For example the multiplication operator requires the two values to be of type `IntVal`, and the resulting type is also an `IntVal`. The return-type of `operate` is `Either String Value`. This allows us to either return the Value or return an appropriate error message when for example the `Div` operator ecounters a division by zero.
The `apply` helper function is used by `eval Call` to call some built-in function. The function takes a function name and a list of evaluated arguments and applies that function to the

arguments if the function exists. If it doesn't exist or the arguments does not match an error is signaled. The different cases are very different because they depend on the function being called. We will go into more details on the built-ins and how apply is used in the particular cases in a following paragraph.
In the expressions `Call FName [Exp]` and `List [Exp]` we have a list of expressions that's supposed to be evaluated from left to right. To do that we have created one additional helper function called `evalExpList`, that takes as input a list of expressions, evaluates every expression to values and packages them into a list of values, it then returns a `Comp [Value]`. This is useful to use in evaluating the `Call FName [Exp]`, which is done with the `apply` helper function which takes already evaluated expressions, that is, a list of values and uses these as its arguments.

Built-in functions There are two built-in functions in Boa, `range` which takes from 1-3 integer arguments and returns an integer range, and `print` which takes any amount of values as arguments and outputs a meaningful representation of the values on one line. As mentioned earlier, the functions are callable by using the Call expression which in turn will call apply with the function name and evaluated arguments.
For `range` we defined 4 cases of apply, where the first 3 cases corresponds to calling `range` with either 1, 2 or 3 arguments and using those arguments as specified in the assignment, and the 4th handles any other case by signaling an EBadArg error. The 3 cases with accepted arguments all uses the utility function `range::Int->Int->Int->[Value]`.
The `print` built-in will accept any type argument in any amount, and when calling apply the arguments are already evaluated, so we know they all evaluate to some value which can be printed. That means there is only one case for 'apply "print"' which will call a utility function `printVal::Either Value [Value]->String` with the argument list, and this function returns a single string with the representations of all the arguments. This string is then outputted before returning NoneVal as specified. `printVal` calls itself recursively if the argument is a list and if the argument is a single value it returns the representation of that value. We handle this by using a Either value as the argument.

eval The `eval` function takes an expression `Exp` and evaluates the expression into a `Comp Value`. There is a function definition of `eval` for every `Exp` declared in `BoaAST.hs`. For example, the call `eval (Const (IntVal 5))`, wraps the value `IntVal 5` in a Comp monad by calling `return (IntVal 5)`. Similar for many declarations of `eval` is that we use the `do`-notation to "extract" the value of an evaluated expression by `do val <- eval x`. This come in handy together with our `evalExpList` helper function when we want to evaulate the expression `Call FName [Exp]`, we want to evaluate the list of expressions into a list of values. Since `evalExpList` returns a `Comp [Value]` and we can get the list of values and call `apply` with the list of values.
Now the evaluation of list comprehensions is a little more exciting. We have an expression on the form `Compr Exp [Qual]`, where `Qual = QFor VName Exp | QIf Exp`. We took a recursive approach to evaluating the list comprehension. We have two branches of recursion. One where we evaluate the list comprehension recursively with one less qualification. The other is when we have a `QFor`-qualification, and we recursively evaluate the list comprehension again, but with one less element in the `QFor`'s list. The base case for evaluating a list comprehension is when there are no qualifications, `Compr e0 []`, then we just return the evaluated expression with `return (eval e0)`. If the list of qualifications is not empty, and lets say we have a `QFor vn expList`, we firstly check if the expression `expList` evaluates to

a `ListVal`. Now we, depending on the case, bind the first element of the list to a variable name and then evaluate the comprehension with the remaining qualifications with `e0_val <- withBinding vn x (eval (Compr e0 qs))`
Then we in the second branch, recursively evaluate the list comprehension to get the rest of the values with
`rest_val <- eval $ Compr e0 ((QFor vn (Const (ListVal xs))) : qs)`.
Then we concatenate the result into one list wich is returned in a Comp.
If we, on our way through the qualifications, encounter a `QIf Exp` we evaluate expression and tests its value with the helper function `truhty`. If the test passes, we continue through the list of qualifications. If not, we stop the recursion and return an empty list.

exec and execute The `exec` function takes a program segment as argument and executes each statement in that program segment. If the statement is a variable definition `SDef` the value is evaluated using `eval` to v and `exec` is called recursively with the remaining program segment as the computation argument of a call to `withBinding` where the evaluated value is bound to the name given in the definition statement. If the statement is an expression `SExp` the expression is evaluated and `exec` is again called recursively with the remainder of the program but this time just as a standalone call (so not as an argument to withBinding). When the program segment is empty of statements the unit value is returned. Since the function returns unit () the point of the function lies in the "side-effects" in the monad, where the environment and output list are getting constructed as the statements are executed and computations are run. In the case of errors the value returned when running the exec computation using `runComp` is the first encountered error, and the output is everything that was outputted up to the point of the error.
The `execute` function is used to run the exec computation with `runComp`, where the initial environment is an empty environment. The function returns a tuple with the output list and an potential error of Maybe type.

2 Code assessment

We used the template `tests/Test.hs` and expanded it with tests as we were implementing the functions in part 2, even though most of the tests were made after the code was done. These tests can be run with the command `stack test`. We made several tests for each function which quickly add up so we have over 100 tests. Still we have not nearly tested the functions exhaustively, so these tests mostly serve to check the functionality of the functions and not as a proof that the functions handle every possible input with grace. We also tested the two example programs in `part2/examples`, by supplying their paths to the executable `boa`, and the output of both tests matched the expected output. Finally we tried uploading the code to onlineTA where every test except 1 is passed both for part 1 and part 2. The one failing test is a test in part 2 of the list comprehension, but the test is pretty strange. The test case is called "eval [... for x in [] ...]" and the program crashes on this test, which means we don't get an expected and actual result but just the following:

```
1 eval [... for x in [] ...]:                    FAIL
2   Exception: forced body without reason
3   CallStack (from HasCallStack):
4     error, called at onlinetatests/Tests.hs:179:29 in main:Main
```

We don't know what is being tested here or what the expected behaviour was supposed to

be so we havn't done anything to fix this test case.
All in all we think the code is decent and not too complicated, with the exception of the list comprehension case of `eval` which got a bit out of hand. There is for example a lot of repeated code in that function, which would imply that we could simplify it, but we didn't have time to look into that. The use of the Comp monad made most function implementations a lot simpler than they would have been without it, passing the output and environment back and forth.
Based on the fact that the code passes all our own tests and the tests on onlineTA (except for that strange one) we feel like our submission meets the requirements of the assigment.

Appendix - Code listings

BoaInterp.hs

```
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, []))
  (Comp c) >>= f = Comp (\env ->
    let (a, l) = c env
      in case a of
        Right x -> let (b, l2) = runComp (f x) env
                   in (b, l++l2)
        Left e ->  (Left e, l))

-- 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 e = Comp (\_ -> (Left e, [] ))

look :: VName -> Comp Value
look vn = Comp ( \env ->
           case env of
             ((x1, x2):xs)
               | x1==vn -> (Right x2, mempty)
               | x1/=vn -> runComp (look vn) xs
             [] -> runComp (abort (EBadVar vn)) []
             _ -> error "impossible case")

withBinding :: VName -> Value -> Comp a -> Comp a
withBinding vn val c = Comp (\env -> let new_env = (vn, val):env
                                     in runComp c new_env)

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

-- Helper functions for interpreter
truthy :: Value -> Bool
truthy NoneVal = False
truthy TrueVal = True
```

```
57  truthy FalseVal = False
58  truthy (IntVal n) = n /= 0
59  truthy (StringVal s) = s /= ""
60  truthy (ListVal l) = (length l) /= 0
61
62  operate :: Op -> Value -> Value -> Either String Value
63  --Plus
64  operate Plus (IntVal v1) (IntVal v2) = Right (IntVal (v1 + v2))
65  -- In python the '+' operator works between strings, strings and ints and
        lists
66  -- which we started by implementing, but then onlineTA complained.
67  -- operate Plus (StringVal s1) (StringVal s2) = Right (StringVal (s1 ++ s2))
68  -- operate Plus (StringVal s) (IntVal i) = Right (StringVal (s ++ (show i)))
69  -- operate Plus (IntVal i) (StringVal s) = Right (StringVal ((show i) ++ s))
70  -- operate Plus (ListVal l1) (ListVal l2) = Right (ListVal (l1 ++ l2))
71  --Minus
72  operate Minus (IntVal v1) (IntVal v2) = Right (IntVal (v1 - v2))
73  --Times
74  operate Times (IntVal v1) (IntVal v2) = Right (IntVal (v1 * v2))
75  --Div
76  operate Div (IntVal v1) (IntVal v2) =
77    if v2 /= 0 then Right (IntVal (v1 `div` v2))
78    else Left "Division by zero."
79  --Mod
80  operate Mod (IntVal v1) (IntVal v2) =
81    if v2 /= 0 then Right (IntVal (v1 `mod` v2))
82    else Left "Modulo by zero."
83  --Eq
84  operate Eq v1 v2 =
85    if v1 == v2 then Right TrueVal
86    else Right FalseVal
87  --Less
88  operate Less (IntVal v1) (IntVal v2) =
89    if v1 < v2 then Right TrueVal
90    else Right FalseVal
91  --Greater
92  operate Greater (IntVal v1) (IntVal v2) =
93    if v1 > v2 then Right TrueVal
94    else Right FalseVal
95  --In
96  operate In v (ListVal l) =
97    if v `elem` l then Right TrueVal
98    else Right FalseVal
99  operate op _ _ = Left (show op ++ ": Operand mismatch.")
100
101 -- Built-in functions
102 -- built-in: range
103 range :: Int -> Int -> Int -> [Value]
104 range n1 n2 n3 =
105   if (n3 > 0 && n1 < n2) || (n3 < 0 && n1 > n2) then
106     (IntVal n1) : (range (n1+n3) n2 n3)
107   else []
108
109 -- built-in: print
110 -- The main function is 'printVal' which is named so because
111 -- the 'print' name is defined in Prelude.
112 printListElements :: [Value] -> String
113 printListElements [] = ""
114 printListElements [x] = (printVal (Left x))
115 printListElements (x:xs) = (printVal (Left x)) ++ ", " ++ (printListElements
```

```
      xs)
116
117 -- aka. print
118 printVal :: Either Value [Value] -> String
119 printVal (Left NoneVal) = "None"
120 printVal (Left TrueVal) = "True"
121 printVal (Left FalseVal) = "False"
122 printVal (Left (IntVal a)) = show a
123 printVal (Left (StringVal s)) = s
124 printVal (Left (ListVal l)) = "[" ++ printListElements l ++ "]"
125 printVal (Right [x]) = (printVal (Left x))
126 printVal (Right (x:xs)) = (printVal (Left x)) ++ " " ++ (printVal (Right xs))
127 printVal (Right []) = ""
128
129 -- apply built-in functions
130 apply :: FName -> [Value] -> Comp Value
131 apply "range" [(IntVal n2)] =
132   return $ ListVal (range 0 n2 1)
133 apply "range" [(IntVal n1), (IntVal n2)] =
134   return $ ListVal (range n1 n2 1)
135 apply "range" [(IntVal n1), (IntVal n2), (IntVal n3)] =
136   return $ ListVal (range n1 n2 n3)
137 apply "range" _ = abort $ EBadArg "invalid arguments for range."
138 apply "print" l = do { output $ printVal (Right l) ; return NoneVal }
139 apply fn _ = abort (EBadFun fn)
140
141 -- helper function that evaluates a list of expressions
142 evalExpList :: [Exp] -> Comp [Value]
143 evalExpList (x:xs) = do
144   a <- eval x
145   rest <- evalExpList xs
146   return $ a : rest
147 evalExpList [] = do return []
148
149 -- Main functions of interpreter
150 eval :: Exp -> Comp Value
151 eval (Const v) = return v
152 eval (Var vn)  = look vn
153 eval (Oper op e1 e2) =
154   do v1 <- eval e1
155      v2 <- eval e2
156      case operate op v1 v2 of
157        Left err -> abort (EBadArg err)
158        Right v  -> return v
159 eval (Not e) = do
160   v <- eval e
161   case v of
162     NoneVal -> return TrueVal
163     FalseVal -> return TrueVal
164     ListVal l   | length l == 0  -> return TrueVal
165     StringVal s | length s == 0  -> return TrueVal
166     IntVal 0 -> return TrueVal
167     _ -> return FalseVal
168 eval (Call f args) = do
169   valList <- evalExpList args
170   apply f valList
171 eval (List el) = do
172   valList <- evalExpList el
173   return (ListVal valList)
174 -- behold
```

```
175 eval (Compr e0 []) = do {a <- eval e0; return $ ListVal [a] }
176 eval (Compr e0 (q:qs)) =
177   case q of
178     QFor vn expList -> do
179       valList <- eval expList
180       case valList of
181         ListVal [] -> eval e0
182         ListVal [x] -> do
183           e0_val <- withBinding vn x (eval (Compr e0 qs))
184           case e0_val of
185             ListVal l -> return $ ListVal l
186             _ -> return $ ListVal [e0_val]
187         ListVal (x:xs) -> do
188           e0_val <- withBinding vn x (eval (Compr e0 qs))
189           rest_val <- eval $ Compr e0 ((QFor vn (Const (ListVal xs))) : qs)
190           case e0_val of
191             ListVal l ->
192               case rest_val of
193                 ListVal ls -> return $ ListVal (l ++ ls)
194                 _ -> return $ ListVal l
195             _ ->
196               case rest_val of
197                 ListVal ls -> return $ ListVal (e0_val : ls)
198                 _ -> return $ ListVal [e0_val]
199         _ -> abort $ EBadArg "Not a list."
200     QIf e -> do
201       cond <- eval e
202       if truthy cond then eval (Compr e0 qs)
203       else return (ListVal [])
204
205 exec :: Program -> Comp ()
206 exec (x:xs) = case x of
207   SDef name exp -> do
208     a <- eval exp
209     withBinding name a (exec xs)
210   SExp exp -> do
211     eval exp
212     exec xs
213 exec [] = do return ()
214
215 execute :: Program -> ([String], Maybe RunError)
216 execute p = let (a, output) = runComp (exec p) []
217             in case a of
218               Left err -> (output, Just err)
219               Right _ -> (output, Nothing)
```

Test.hs

```
1  -- Skeleton test suite using Tasty.
2  -- Fell free to modify or replace anything in this file
3
4  import BoaAST
5  import BoaInterp
6
7  import Test.Tasty
8  import Test.Tasty.HUnit
9
10 main :: IO ()
11 main = defaultMain $ localOption (mkTimeout 1000000) tests
12
13 tests :: TestTree
14 tests =
15   testGroup "All tests"
16   [testGroup "Monad operators"
17   [ -- Comp return and bind
18     testCase "compBase1" $
19       (runComp (return ()) [])
20       @?= (Right (), []),
21     testCase "compBase2" $
22       (runComp (Comp (\_ -> (Right (), ["some output"]))) [])
23       @?= (Right (), ["some output"]),
24     testCase "compBase3" $
25       (runComp (do { x <- return (); return x}) [])
26       @?= (Right (), []),
27     testCase "compBase4" $
28       (runComp (do { x <- return "Hello"; return x}) [])
29       @?= (Right "Hello", []),
30     -- output
31      testCase "output1" $
32       (runComp (do { output "test output"; return () }) [])
33       @?= (Right (), ["test output"]),
34     testCase "output2" $
35       (runComp (do { output "test1"; output "test2"; return () }) [])
36       @?= (Right (), ["test1", "test2"]),
37     -- abort
38     testCase "abort1" $
39       ((runComp (abort (EBadVar "x")) [])
40         :: (Either RunError String, [String]))
41       @?= (Left (EBadVar "x"), []),
42     testCase "abort2" $
43       ((runComp (do { abort (EBadFun "f");
44                       output "no errors?";
45                       return ()}) [])
46         :: (Either RunError (), [String]))
47       @?= (Left (EBadFun "f"), []),
48     testCase "abort3" $
49       ((runComp (do { output "before crash";
50                       abort (EBadArg "a");
51                       output "after crash";
52                       return ()}) [])
53         :: (Either RunError (), [String]))
54       @?= (Left (EBadArg "a"), ["before crash"]),
55     -- look
56     testCase "look1" $
57       (runComp (look "a") [("a", IntVal 1)])
58       @?= (Right (IntVal 1), []),
```

```
59      testCase "look2" $
60        (runComp (look "a") [("b", IntVal 2), ("a", IntVal 1)])
61        @?= (Right (IntVal 1), []),
62      testCase "look3" $
63        (runComp (look "a") [("a", IntVal 3), ("b", IntVal 2), ("a", IntVal 1)
              ])
64        @?= (Right (IntVal 3), []),
65      testCase "look4" $
66        (runComp (look "a") [])
67        @?= (Left (EBadVar "a"), []),
68      testCase "look5" $
69        (runComp (do { a <- look "a"; output (show a); return () })
70          [("a", IntVal 1)])
71        @?= (Right (), ["IntVal 1"]),
72      -- withBinding
73      testCase "withBinding1" $
74        (runComp (withBinding "a" (IntVal 1) (look "a")) [])
75        @?= (Right (IntVal 1), []),
76      testCase "withBinding2" $
77        (runComp (withBinding "a" (IntVal 2) (look "a")) [("a", IntVal 1)])
78        @?= (Right (IntVal 2), []),
79      testCase "withBinding3" $
80        (runComp (withBinding "a" (IntVal 2) (look "b")) [])
81        @?= (Left (EBadVar "b"), []),
82      testCase "withBinding4" $
83        (runComp (withBinding "a" (IntVal 2) (look "b"))
84          [("b", StringVal "Maxwell")])
85        @?= (Right (StringVal "Maxwell"), []),
86      testCase "withBinding5" $
87        (runComp (withBinding "a" (StringVal "Oh, ") ( do
88          a <- look "a"
89          b <- look "b"
90          case (a, b) of
91            (StringVal s1, StringVal s2) -> return $ s1 ++ s2
92            _ -> return ""
93          )) [("b", StringVal "Darlin'")])
94        @?= (Right "Oh, Darlin'", [])],
95
96    testGroup "Helper functions truthy and operate"
97      [
98      -- truthy
99      testCase "truthy1" $ truthy NoneVal @?= False,
100     testCase "truthy2" $ truthy TrueVal @?= True,
101     testCase "truthy3" $ truthy FalseVal @?= False,
102     testCase "truthy4" $ truthy (IntVal 0) @?= False,
103     testCase "truthy5" $ truthy (IntVal 1) @?= True,
104     testCase "truthy6" $ truthy (IntVal (-1)) @?= True,
105     testCase "truthy7" $ truthy (StringVal "") @?= False,
106     testCase "truthy7" $ truthy (StringVal "hey") @?= True,
107     testCase "truthy7" $ truthy (ListVal []) @?= False,
108     testCase "truthy7" $ truthy (ListVal [IntVal 1]) @?= True,
109     -- operate
110     -- operate Plus
111     testCase "operatePlus1" $
112       operate Plus (IntVal 1) (IntVal 2)
113       @?= Right (IntVal 3),
114     testCase "operatePlus2" $
115       operate Plus NoneVal (IntVal 2)
116       @?= Left ("Plus: Operand mismatch."),
117     testCase "operatePlus3" $
```

```
118         operate Plus (FalseVal) (TrueVal)
119         @?= Left ("Plus: Operand mismatch."),
120       -- operate Minus
121       testCase "operateMinus1" $
122         operate Minus (IntVal 2) (IntVal 1)
123         @?= Right (IntVal 1),
124       testCase "operateMinus2" $
125         operate Minus (IntVal (-99)) (IntVal (-100))
126         @?= Right (IntVal 1),
127       testCase "operateMinus3" $
128         operate Minus (NoneVal) (IntVal 0)
129         @?= Left ("Minus: Operand mismatch."),
130       -- operate Times
131       testCase "operateTimes1" $
132         operate Times (IntVal 2) (IntVal 5)
133         @?= Right (IntVal 10),
134       testCase "operateTimes2" $
135         operate Times (IntVal 0) (IntVal 5)
136         @?= Right (IntVal 0),
137       testCase "operateTimes3" $
138         operate Times (IntVal (-1)) (IntVal 10)
139         @?= Right (IntVal (-10)),
140       testCase "operateTimes4" $
141         operate Times (IntVal (-2)) (IntVal (-2))
142         @?= Right (IntVal (4)),
143       testCase "operateTimes5" $
144         operate Times NoneVal (IntVal (-2))
145         @?= Left ("Times: Operand mismatch."),
146       -- operate Div
147       testCase "operateDiv1" $
148         operate Div (IntVal 4) (IntVal 2)
149         @?= Right (IntVal 2),
150       testCase "operateDiv2" $
151         operate Div (IntVal 5) (IntVal 2)
152         @?= Right (IntVal 2),
153       testCase "operateDiv3" $
154         operate Div (IntVal 0) (IntVal 2)
155         @?= Right (IntVal 0),
156       testCase "operateDiv4" $
157         operate Div (IntVal 2) (IntVal 0)
158         @?= Left ("Division by zero."),
159       -- operate Mod
160       testCase "operateMod1" $
161         operate Mod (IntVal 4) (IntVal 2)
162         @?= Right (IntVal 0),
163       testCase "operateMod2" $
164         operate Mod (IntVal 5) (IntVal 2)
165         @?= Right (IntVal 1),
166       testCase "operateMod3" $
167         operate Mod (IntVal 0) (IntVal 2)
168         @?= Right (IntVal 0),
169       testCase "operateMod4" $
170         operate Mod (IntVal 2) (IntVal 0)
171         @?= Left ("Modulo by zero."),
172       -- operate Eq
173       testCase "operateEq1" $
174         operate Eq (IntVal 2) (IntVal 2)
175         @?= Right (TrueVal),
176       testCase "operateEq2" $
177         operate Eq (IntVal 1) (IntVal 2)
```

```
178         @?= Right (FalseVal),
179       testCase "operateEq3" $
180         operate Eq (IntVal 1) (StringVal "one")
181         @?= Right (FalseVal),
182       testCase "operateEq4" $
183         operate Eq (StringVal "one") (StringVal "one")
184         @?= Right (TrueVal),
185       testCase "operateEq5" $
186         operate Eq (StringVal "one") (StringVal "two")
187         @?= Right (FalseVal),
188       testCase "operateEq6" $
189         operate Eq (ListVal [IntVal 2]) (IntVal 2)
190         @?= Right (FalseVal),
191       testCase "operateEq7" $
192         operate Eq (ListVal [IntVal 1, IntVal 2])
193           (ListVal [IntVal 1, IntVal 2])
194         @?= Right (TrueVal),
195       testCase "operateEq8" $
196         operate Eq NoneVal NoneVal
197         @?= Right (TrueVal),
198       testCase "operateEq9" $
199         operate Eq TrueVal FalseVal
200         @?= Right (FalseVal),
201       testCase "operateEq9" $
202         operate Eq TrueVal TrueVal
203         @?= Right (TrueVal),
204       -- operate Less
205       testCase "operateLess1" $
206         operate Less (IntVal 2) (IntVal 2)
207         @?= Right (FalseVal),
208       testCase "operateLess2" $
209         operate Less (IntVal 1) (IntVal 2)
210         @?= Right (TrueVal),
211       testCase "operateLess3" $
212         operate Less (IntVal 2) (IntVal 1)
213         @?= Right (FalseVal),
214       testCase "operateLess4" $
215         operate Less (IntVal (-10)) (IntVal 1)
216         @?= Right (TrueVal),
217       testCase "operateLess5" $
218         operate Less (IntVal 1) (StringVal "1")
219         @?= Left ("Less: Operand mismatch."),
220       -- operate Greater
221       testCase "operateGreater1" $
222         operate Greater (IntVal 2) (IntVal 2)
223         @?= Right (FalseVal),
224       testCase "operateGreater2" $
225         operate Greater (IntVal 1) (IntVal 2)
226         @?= Right (FalseVal),
227       testCase "operateGreater3" $
228         operate Greater (IntVal 2) (IntVal 1)
229         @?= Right (TrueVal),
230       testCase "operateGreater4" $
231         operate Greater (IntVal (-10)) (IntVal 1)
232         @?= Right (FalseVal),
233       testCase "operateGreater5" $
234         operate Greater (IntVal 1) (StringVal "1")
235         @?= Left ("Greater: Operand mismatch."),
236       -- operate In
237       testCase "operateIn1" $
```

```
238         operate In (IntVal 2) (ListVal [IntVal 1, IntVal 2, IntVal 3])
239         @?= Right (TrueVal),
240       testCase "operateIn2" $
241         operate In (IntVal 4) (ListVal [IntVal 1, IntVal 2, IntVal 3])
242         @?= Right (FalseVal),
243       testCase "operateIn3" $
244         operate In (StringVal "beans")
245           (ListVal [IntVal 1, IntVal 2, StringVal "beans", IntVal 3])
246         @?= Right (TrueVal),
247       testCase "operateIn4" $
248         operate In (StringVal "beans")
249           (ListVal [StringVal "cool beans"])
250         @?= Right (FalseVal),
251       testCase "operateIn5" $
252         operate In (TrueVal)
253           (ListVal [NoneVal, FalseVal, IntVal 2, TrueVal, NoneVal])
254         @?= Right (TrueVal),
255       testCase "operateIn6" $
256         operate In (ListVal [TrueVal])
257           (ListVal [NoneVal, FalseVal, IntVal 2, TrueVal, NoneVal])
258         @?= Right (FalseVal),
259       testCase "operateIn7" $
260         operate In (ListVal [TrueVal])
261           (ListVal [NoneVal, FalseVal, IntVal 2, ListVal [TrueVal]])
262         @?= Right (TrueVal),
263       testCase "operateIn8" $
264         operate In TrueVal FalseVal
265         @?= Left ("In: Operand mismatch.")
266       ],
267     testGroup "apply (and built-ins)"
268       [ -- apply range
269       testCase "range1" $
270         runComp (apply "range" [IntVal 3]) []
271         @?= (Right (ListVal [IntVal 0, IntVal 1, IntVal 2]), []),
272       testCase "range2" $
273         runComp (apply "range" [IntVal 0, IntVal 3]) []
274         @?= (Right (ListVal [IntVal 0, IntVal 1, IntVal 2]), []),
275       testCase "range3" $
276         runComp (apply "range" [IntVal 0, IntVal 3, IntVal 1]) []
277         @?= (Right (ListVal [IntVal 0, IntVal 1, IntVal 2]), []),
278       testCase "range4" $
279         runComp (apply "range" [IntVal 0, IntVal 3, IntVal 2]) []
280         @?= (Right (ListVal [IntVal 0, IntVal 2]), []),
281       testCase "range5" $
282         runComp (apply "range" [IntVal (-2), IntVal 3, IntVal 2]) []
283         @?= (Right (ListVal [IntVal (-2), IntVal 0, IntVal 2]), []),
284       testCase "range6" $
285         runComp (apply "range" [IntVal 0]) []
286         @?= (Right (ListVal []), []),
287       testCase "range7" $
288         runComp (apply "range" [IntVal (-1)]) []
289         @?= (Right (ListVal []), []),
290       testCase "range8" $
291         runComp (apply "range" [IntVal 3, IntVal 0]) []
292         @?= (Right (ListVal []), []),
293       testCase "range9" $
294         runComp (apply "range" [IntVal 3, IntVal 0, IntVal (-1)]) []
295         @?= (Right (ListVal [IntVal 3, IntVal 2, IntVal 1]), []),
296       testCase "range10" $
297         runComp (apply "range" [IntVal 3, IntVal 0, IntVal 1]) []
```

```
298         @?= (Right (ListVal []), []),
299       testCase "range11" $
300         runComp (apply "range" [StringVal "hello"]) []
301         @?= (Left (EBadArg "invalid arguments for range."), []),
302       testCase "range12" $
303         runComp (apply "range" [IntVal 3, StringVal "hello"]) []
304         @?= (Left (EBadArg "invalid arguments for range."), []),
305       -- apply print
306       testCase "print1" $
307         runComp (apply "print" [StringVal "Hello world!"]) []
308         @?= (Right NoneVal, ["Hello world!"]),
309       testCase "print2" $
310         runComp (apply "print" [StringVal "Hello", StringVal "world!"]) []
311         @?= (Right NoneVal, ["Hello world!"]),
312       testCase "print3" $
313         runComp (apply "print" [StringVal "Power: >", IntVal 9000]) []
314         @?= (Right NoneVal, ["Power: > 9000"]),
315       testCase "print4" $
316         runComp (apply "print" [IntVal 42, StringVal "foo",
317           ListVal [TrueVal, ListVal []], IntVal (-1)]) []
318         @?= (Right NoneVal, ["42 foo [True, []] -1"]),
319       testCase "print5" $
320         runComp (apply "print" [ListVal [IntVal 1, IntVal 2]]) []
321         @?= (Right NoneVal, ["[1, 2]"])
322       ],
323     testGroup "eval"
324       [ -- eval Const
325       testCase "evalConst1" $
326         runComp (eval (Const (IntVal 1))) []
327         @?= (Right (IntVal 1), []),
328       testCase "evalConst2" $
329         runComp (eval (Const (NoneVal))) []
330         @?= (Right NoneVal, []),
331       -- eval Var
332       testCase "evalVar1" $
333         runComp (eval (Var ("x"))) testEnv1
334         @?= (Right (IntVal 2), []),
335       testCase "evalVar2" $
336         runComp (eval (Var ("y"))) testEnv1
337         @?= (Right (IntVal 10), []),
338       testCase "evalVar3" $
339         runComp (eval (Var ("name"))) testEnv1
340         @?= (Right (StringVal "Jim"), []),
341       testCase "evalVar4" $
342         runComp (eval (Var ("w"))) testEnv1
343         @?= (Left (EBadVar "w"), []),
344       -- eval Oper
345       testCase "evalOper1" $
346         runComp (eval (Oper Plus (Const (IntVal 1)) (Const (IntVal 1)))) []
347         @?= (Right (IntVal 2), []),
348       testCase "evalOper2" $
349         runComp (eval (Oper Times (Const (IntVal 2)) (Const (IntVal 2)))) []
350         @?= (Right (IntVal 4), []),
351       testCase "evalOper3" $
352         runComp (eval (Oper Times (Var "x") (Const (IntVal 2)))) testEnv1
353         @?= (Right (IntVal 4), []),
354       testCase "evalOper4" $
355         runComp (eval (Oper Div (Var "z") (Var "x"))) testEnv1
356         @?= (Right (IntVal (-3)), []),
357       testCase "evalOper5" $
```

```
358         runComp (eval (Oper Mod (Var "z") (Var "x"))) testEnv1
359         @?= (Right (IntVal 1), []),
360       testCase "evalOper6" $
361         runComp (eval (Oper Eq (Var "name")
362           (Const (StringVal "Jim")))) testEnv1
363         @?= (Right (TrueVal), []),
364       testCase "evalOper7" $
365         runComp (eval (Oper Less (Var "z") (Var "x"))) testEnv1
366         @?= (Right (TrueVal), []),
367       testCase "evalOper8" $
368         runComp (eval (Oper Greater (Var "y") (Var "x"))) testEnv1
369         @?= (Right (TrueVal), []),
370       testCase "evalOper9" $
371         runComp (eval (Oper Greater (Var "y") (Var "x"))) testEnv1
372         @?= (Right (TrueVal), []),
373       testCase "evalOper10" $
374         runComp (eval (Oper In (Var "name")
375           (List [Var "x", Const (StringVal "Jim")]))) testEnv1
376         @?= (Right (TrueVal), []),
377       testCase "evalOper11" $
378         runComp (eval (Oper Plus (Const (IntVal 1)) (Const NoneVal))) []
379         @?= (Left (EBadArg "Plus: Operand mismatch."), []),
380       -- eval Not
381       testCase "evalNot1" $
382         runComp (eval (Not (Const NoneVal))) []
383         @?= (Right TrueVal, []),
384       testCase "evalNot2" $
385         runComp (eval (Not (Var "x"))) testEnv1
386         @?= (Right FalseVal, []),
387       testCase "evalNot3" $
388         runComp (eval (Not (Const (StringVal "")))) testEnv1
389         @?= (Right TrueVal, []),
390       testCase "evalNot4" $
391         runComp (eval (Not (Var "name"))) testEnv1
392         @?= (Right FalseVal, []),
393       testCase "evalNot5" $
394         runComp (eval (Not (List []))) testEnv1
395         @?= (Right TrueVal, []),
396       -- eval Call
397       testCase "evalCall-range1" $
398         runComp (eval (Call "range" [(Const (IntVal 3))] )) []
399         @?= (Right (ListVal [IntVal 0, IntVal 1, IntVal 2]), []),
400         testCase "evalCall-range2" $
401         runComp (eval (Call "range" [Const (IntVal 0), Const (IntVal 3)] )) []
402         @?= (Right (ListVal [IntVal 0, IntVal 1, IntVal 2]), []),
403       testCase "evalCall-range3" $
404         runComp (eval (Call "range" [Const (IntVal 0), Const (IntVal 3),
405                                      Const (IntVal 1)] )) []
406         @?= (Right (ListVal [IntVal 0, IntVal 1, IntVal 2]), []),
407       testCase "evalCall-range4" $
408         runComp (eval (Call "range" [Const (IntVal 0),
409                                      Const (IntVal 3), Const (IntVal 2)])) []
410         @?= (Right (ListVal [IntVal 0, IntVal 2]), []),
411       testCase "evalCall-range5" $
412         runComp (eval (Call "range" [Const (IntVal (-2)),Const (IntVal 3),
413                                      Const (IntVal 2)])) []
414         @?= (Right (ListVal [IntVal (-2), IntVal 0, IntVal 2]), []),
415       testCase "evalCall-range6" $
416         runComp (eval (Call "range" [Const (IntVal 0)])) []
417         @?= (Right (ListVal []), []),
```

```
418       testCase "evalCall-range7" $
419         runComp (eval (Call "range" [Const (IntVal (-1))])) []
420         @?= (Right (ListVal []), []),
421       testCase "evalCall-range8" $
422         runComp (eval (Call "range" [Const (IntVal 3),Const (IntVal 0)])) []
423         @?= (Right (ListVal []), []),
424       testCase "evalCall-range9" $
425         runComp (eval (Call "range" [Const (IntVal 3),Const (IntVal 0),
426                                      Const (IntVal (-1))])) []
427         @?= (Right (ListVal [IntVal 3, IntVal 2, IntVal 1]), []),
428       testCase "evalCall-range10" $
429         runComp (eval (Call "range" [Const (IntVal 3),Const (IntVal 0),
430                                      Const (IntVal 1)])) []
431         @?= (Right (ListVal []), []),
432       testCase "evalCall-range11" $
433         runComp (eval (Call "range" [Const (StringVal "hello")])) []
434         @?= (Left (EBadArg "invalid arguments for range."), []),
435       testCase "evalCall-range12" $
436         runComp (eval (Call "range" [Const (IntVal 3),
437                                      Const (StringVal "hello")])) []
438         @?= (Left (EBadArg "invalid arguments for range."), []),
439         testCase "evalCall-print1" $
440         runComp (eval (Call "print" [Const (StringVal "Hello world!")])) []
441         @?= (Right NoneVal, ["Hello world!"]),
442       testCase "evalCall-print2" $
443         runComp (eval (Call "print" [Const (StringVal "Hello"),
444                                      Const (StringVal "world!")])) []
445         @?= (Right NoneVal, ["Hello world!"]),
446       testCase "evalCall-print3" $
447         runComp (eval (Call "print" [Const (StringVal "Power: >"),
448                                      Const (IntVal 9000) ])) []
449         @?= (Right NoneVal, ["Power: > 9000"]),
450       testCase "evalCall-print4" $
451         runComp (eval (Call "print" [Const (IntVal 42), Const (StringVal "foo")
              ,
452                                      Const (ListVal [TrueVal, ListVal []]),
453                                      Const (IntVal (-1))])) []
454         @?= (Right NoneVal, ["42 foo [True, []] -1"]),
455
456         --testCase "evalCall-print5" $
457       --   runComp (eval (Call "print" [Const (ListVal [IntVal 1,
458       --                                                IntVal 2])])) []
459       --   @?= (Right NoneVal, ["[1, 2]"])
460       -- eval List
461       testCase "evalList1" $
462         runComp (eval (List [Var "x", Var "y", Var "z"])) testEnv1
463         @?= (Right (ListVal [IntVal 2, IntVal 10, IntVal (-5)]), []),
464       testCase "evalList2" $
465         runComp (eval (List [])) []
466         @?= (Right (ListVal []), []),
467       testCase "evalList3" $
468         runComp (eval (List [List [Var "x"], List [Var "y"]])) testEnv1
469         @?= (Right (ListVal [ListVal [IntVal 2],ListVal [IntVal 10]]), []),
470       testCase "evalList4" $
471         runComp (eval (List [Var "w"])) testEnv1
472         @?= (Left (EBadVar "w"), []),
473       -- eval Compr
474       testCase "evalCompr1" $
475         runComp (eval (Compr (Const (IntVal 42)) [])) []
476         @?= (Right (ListVal [IntVal 42]), []),
```

```
477       testCase "evalCompr2" $
478       runComp (eval (Compr (Const (IntVal 42)) [QIf (Const FalseVal)])) []
479       @?= (Right (ListVal []), []),
480       testCase "evalCompr3" $
481         runComp (eval (Compr (Var "x") [QFor "x" (List [Const (IntVal 1),
482                                                         Const (IntVal 2)]) ]))
                                                              []
483         @?= (Right (ListVal [IntVal 1, IntVal 2]), []),
484       testCase "evalCompr4" $
485       runComp (eval (Compr (Const (IntVal 10) )
486                     [QFor "x" (List [Const (IntVal 1), Const (IntVal 2)]) ]))
                           []
487       @?= (Right (ListVal [IntVal 10, IntVal 10]), []),
488       testCase "evalCompr5" $
489         runComp (eval (Compr (Var "x")
490                           [QFor "x" (List [Const (IntVal 1), Const (IntVal 2)]),
491                            QIf (Oper Greater (Var "x") (Const (IntVal 1) )) ]))
                                 []
492         @?= (Right (ListVal [IntVal 2]), []),
493       testCase "evalCompr6" $
494         runComp (eval (Compr (Var "x")
495                           [QFor "x" (List [Const (IntVal 0)]),
496                           QIf (Not (Var "x")) ])) []
497         @?= (Right (ListVal [IntVal 0]), []),
498       testCase "evalCompr7" $
499       runComp (eval (Compr (Var "y")
500                         [QFor "x" (List [Const (IntVal 0),Const (IntVal 1)])]))
                               []
501       @?= (Left (EBadVar "y"),[]),
502       testCase "evalCompr8" $
503       runComp (eval (Compr (Var "y")
504                         [QIf (Const (FalseVal)) ,
505                          QFor "x" (List [Const (IntVal 0),Const (IntVal 1)])]))
                                 []
506       @?= (Right (ListVal []), []),
507       testCase "evalCompr9" $
508       runComp (eval (Compr (Oper Plus (Var "y") (Var "x"))
509                         [QFor "x" (List [Const (IntVal 1),Const (IntVal 2)]),
510                          QFor "y" (List [Const (IntVal 3), Const (IntVal 4)])]))
                                 []
511       @?= (Right (ListVal [IntVal 4, IntVal 5, IntVal 5, IntVal 6]), [])
512       ],
513     testGroup "exec and execute"
514       [-- exec
515       testCase "exec1" $
516         runComp (exec []) testEnv1
517         @?= (Right (), []),
518       testCase "exec2" $
519         runComp (exec [SExp (Call "print"
520           [Var "name", Const (StringVal "is my name.")])]) testEnv1
521         @?= (Right (), ["Jim is my name."]),
522       testCase "exec3" $
523         runComp (exec [
524           SDef "height" (Const (IntVal 20)), SExp (Call "print"
525             [Const (StringVal "The tower is"), Var "height",
526             Const (StringVal "meters tall.")])]) testEnv1
527         @?= (Right (), ["The tower is 20 meters tall."]),
528       testCase "exec4" $
529         runComp (exec beforeAfter) []
530         @?= (Left (EBadVar "w"), ["Before crash."]),
```

```
531       -- execute
532       testCase "execute1" $
533         execute beforeAfter @?= (["Before crash."], Just (EBadVar "w")),
534       testCase "execute funCompr" $
535         execute funCompr @?= funComprOut,
536       testCase "execute crash.boa" $
537         execute crashAST @?= crashOut
538       ]]
539     where
540       testEnv1 = [("x", IntVal 2), ("y", IntVal 10), ("z", IntVal (-5)),
541                   ("name", StringVal "Jim"), ("y", IntVal 50)]
542       crashAST = [SExp (Call "print" [Oper Plus (Const (IntVal 2))
543                                                 (Const (IntVal 2))]),
544                   SExp (Var "hello")]
545       crashOut = (["4"], Just (EBadVar "hello"))
546       beforeAfter = [ SExp (Call "print" [Const (StringVal "Before crash.")]),
547                       SExp (Var "w"),
548                       SExp (Call "print" [Const (StringVal "After crash.")])]
549       funCompr = [ SDef "res" (Compr (Oper Plus (Var "x") (Var "y"))
550                           [(QFor "x" (Call "range" [(Const (IntVal 6))])),
551                           (QFor "y" (List [Oper Times (Var "x")
552                                                       (Const (IntVal 2))]))]),
553                    SExp (Call "print" [(Var "res")])]
554       funComprOut = (["[0, 3, 6, 9, 12, 15]"], Nothing)
```