KØBENHAVNS UNIVERSITET

Home Assignment 2

Boa Interpreter-Haskell

qbp758 / zxf339

AP22 Assignment 2 Group 17

Date: 23. september 2022

Part 1: Explanation about A Boa interpreter

1. How our *return* and >>= functions work

Regarding the *return* function, it takes any value and lifts it to a *Monad*. In this question, the new type *Comp* is for Boa computations, and *Comp* is an instance of *Monad*. So, we support the *return* function to take any value to a *CompMonad* and pass the value to the *Right* of *Either* in the *CompMonad*. Thus, when the *return* function is called, it will get a *CompMonad* of that value. The code is as follows:

```
-- return :: Monad m => a -> m a
return a = Comp $ \_ -> (Right a, [])
```

The bind function (>>=) supports chain operation. In this question, first parse out the left Monad m, if there is an error then report it and terminate, if there is a correct value and pass it to the right f function and parse $f(a)$, then return the result and combine the two parsed out lists in the result. *runComp* is used to parse a *CompMonad* and *mappend*(<>) is used for combining out lists. The code is follows.

```
--  (>>=) :: m a -> (a -> m b) -> m b
m >>= f = Comp $ \e -> case runComp m e of -- parse the left Comp Monad
                            (Left err, s) -> (Left err, s)
                            (Right a, s) ->
                              case runComp (f a) e of -- parse f(a)
                                (Left err, s') -> (Left err, s <> s')-- combine out lists
                                (Right a', s'') -> (Right a', s <> s'')-- combine out lists
```

2. How we implemented Boa's list comprehensions

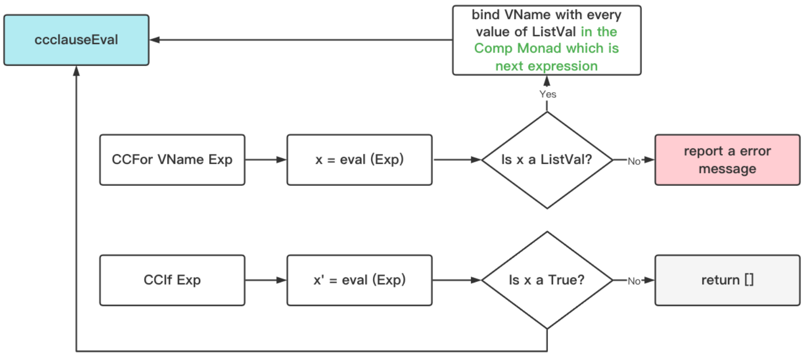
We use a helper function to implemented list comprehensions, the definition of the helper function as follows:

```
ccclauseEval :: Exp -> [CClause] -> Comp [Value]
```

Firstly, for the *CCFor* expressions:

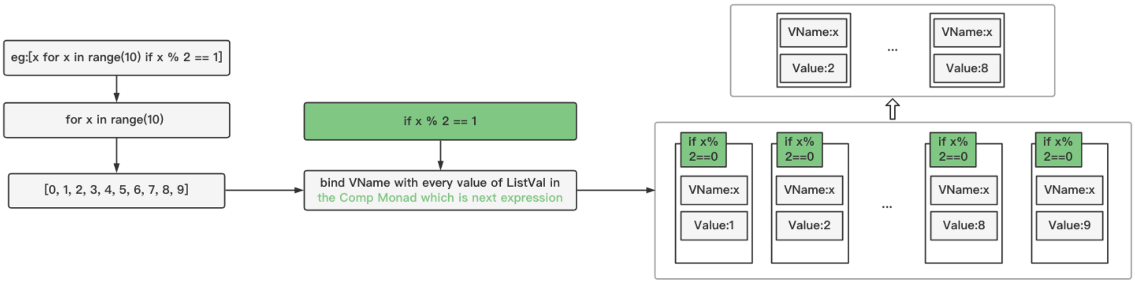
- First, use the *eval* function to parse the e expression (the second part of CCFor).
- If $eval(e)$ gets a non-ListVal will report a error.
- If $eval(e)$ gets a ListVal(noted l), then use the list comprehensions feature of Haskell to handle every value of the ListVal l, bind *VName*(first part of CCFor) with the value of l in a Comp Monad which is next expression.
- At this point, the next expression maybe a *CCIf* expression, so, we bind every value of a ListValin this *CCIf* Comp Monad, namely, we get the new ListVal that every value of it must match the *CCif* condition.

Secondly, for the *CCIf* expressions, use the *eval* function to parse the e expression(the right part of CCIf) to get a value(noted x). If x is true, then recursively parse the next expression, otherwise, return empty list([ ]).



Let’s draw a picture to explain an example:

```
--  [x for x in range(10) if x % 2 == 1]
Compr (Var "x") [CCFor "x" (Call "range" [Const (IntVal 10)]),
                                      CCIf (Oper Eq (Oper Mod (Var "x")
                                                              (Const (IntVal 2)))
                                                    (Const (IntVal 1)))]
```



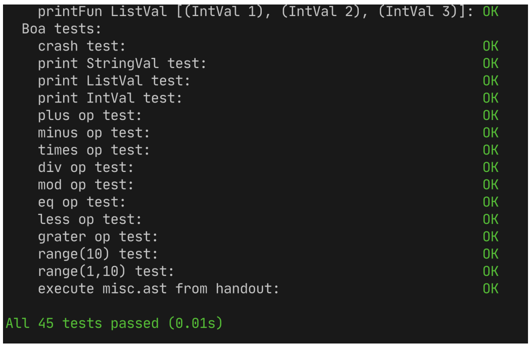
Part 2: Assessment of the Quality of Our code

A. Completeness

We have completed all the questions, although we spent a bunch of time.

B. Correctness

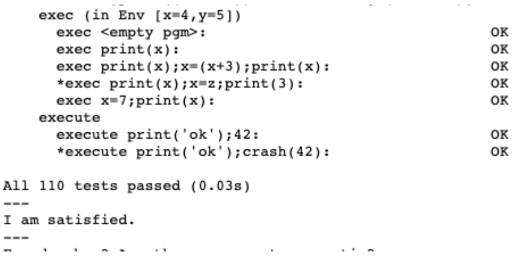
Our code has passed both the unit-testing and the onlineTA-testing.



```
    printFun ListVal [(IntVal 1), (IntVal 2), (IntVal 3)]: OK
  Boa tests:
    crash test:                                          OK
    print StringVal test:                                OK
    print ListVal test:                                  OK
    print IntVal test:                                   OK
    plus op test:                                        OK
    minus op test:                                       OK
    times op test:                                       OK
    div op test:                                         OK
    mod op test:                                         OK
    eq op test:                                          OK
    less op test:                                        OK
    grater op test:                                      OK
    range(10) test:                                      OK
    range(1,10) test:                                    OK
    execute misc.ast from handout:                       OK

All 45 tests passed (0.01s)
```

(a) Result of Unit Testing



```
    exec (in Env [x=4,y=5])
      exec <empty pgm>:                                      OK
      exec print(x):                                         OK
      exec print(x);x=(x+3);print(x):                        OK
      *exec print(x);x=z;print(3):                           OK
      exec x=7;print(x):                                     OK
    execute
      execute print('ok');42:                                OK
      *execute print('ok');crash(42):                        OK

All 110 tests passed (0.03s)
---
I am satisfied.
---
```

(b) Result of OnlineTA Testing

C. Efficiency

This time, in addition to trying our best to improve the efficiency of code operation, we also made efforts to improve space usage. It turns out that our result is better than last time.

D. Robustness

The Boa interpreter we designed can cope with errors and erroneous input. We have implemented some errors processing, for example, in the ‘div‘ ‘Mod‘ operation, and it will report an error message when the dividend number or modulus equals zero; For erroneous input, in the ‘in‘ and ‘list comprehension‘ operations, if the parameter is non-list, it will report an error message.

E. Maintainability

We both think the maintainability of our code is quite good. As we added enough common code to improve the readability of the code and abstracted functions to make our code logic clarified.