Advanced Programming Assignment 3 : A Boa Parser Report

AP22 Assignment 3 Group 60
KRW521, CPJ395

September 29, 2022

1 Design and Implementation

1.1 Design and skeleton of the program

Here we want to introduce our design's idea. In the syntax hierarchy of Boa, from abstract to concrete, the types are ordered in "**Program, Stmts, Stmt, Expr**" where **Expr** contains the concrete expressions like **numConst, stringConst, Oper** and so on. Likewise, the target type in Haskell language also has the similar syntax hierarchy, from abstract to concrete, the types are ordered in "**Program, Stmt, Exp**" where **Exp** contains the concrete expressions like **Const Value, Var VName, Oper** and so on. While coding the parsing program using **ReadP**(the parser), we create **ReadP** for different type: **ReadP Program, ReadP [Stmt], ReadP Stmt, ReadP Exp**. Basically, our idea is to use functions to combine the concrete types' **ReadP**s and make them become the **ReadP** of abstract types.

Concretely, for each of the abstract type: **Program,** [**Stmt**], and **Stmt**, we allocate one function in which a **ReadP** of a more concrete type is used to generate a **ReadP** for it(e.g. we use **ReadP Stmt** in function **rpStmts** to **generate ReadP** [**Stmt**]). But when we get into the type **Exp**, we find it difficult to use only one function to generate **ReadP** for **Exp**, for it will cause several problems like errors in the priority of operators bringing us difficulties in combinator-based parsing. So here we change the grammar in Exp (divide **Exp** into several classes in grammar), that is, set **Oper Op Exp Exp** as the **Exp** that has the highest parsing priority, in which relation operation (like <, >, ==...) **Exp** parse first (we use the function **outerOperExp** that uses the result from the function **intermediateOperExp** to generate **ReadP Exp** for them), then the **Plus** and **Minus** operation (we use the function **intermediateOperExp** that uses the result from the function **innerOperExp** to generate **ReadP Exp** for them), next are **Times, Div** and **Mod** (we use the function **innerOperExp** that uses the result from the function **concreteOperExp** to generate **ReadP Exp** for them), and then are the basic elements like **List, Const, Call**... (we use the function **concreteOperExp** that uses function **operExp** which is also the function **outerOperExp** to generate **ReadP Exp** for them)

For the ambiguity like $notnotx < 3$, every time the program detect "**not** " in function outerOperExp , it will recursively check the expressions that follow it, and return **Not Exp**. So this kind of ambiguity can be solved. For the relational operation like $x > y > z$, every time the > matches two **Exp** from function **intermediateOperExp** it will soon return **Oper Greater**

e1 e2 making it have no chance to match the remaining z, so this kind of ambiguity can be solved. For left-recursion in **Plus Minus** and **Times Div Mod**, we use Helper function (take the Helper function for dealing with the left-recursion in Plus Minus as an example):

```
intermediateOperExp :: ReadP Exp
intermediateOperExp = (do
                e1 <- innerOperExp
                intermediateOperExpHelper e1)
        <++ innerOperExp

-- Using helper funciton to realize left association
intermediateOperExpHelper :: Exp -> ReadP Exp
intermediateOperExpHelper e1 = ( do
                token (char '+')
                e2 <- innerOperExp
                intermediateOperExpHelper (Oper Plus e1 e2))
        <++ ( do
                token (char '-')
                e2 <- innerOperExp
                intermediateOperExpHelper (Oper Minus e1 e2))
        <++ return e1
```

After accepting the left **Exp**, the program will soon call the Helper function with the left **Exp**, then the helper function will search that if there is another + or − to decide whether it should return that left **Exp** or calling itself recursively according to the operator it meets. It goes the same way in **Times Div** and **Mod**. By doing so, the left-recursion can be solved.

As for the biased combinator $<++$, we would like to use it in the functions that generate **ReadP** for types from abstract to concrete (like functions **outerOperExp, intermediateOperExp, innerOperExp, concreteOperExp** and so on). In our opinions, parsing is a process that tries to match the pattern written in the parser. If not matched, then the consumed char of the input string will be resumed by backtracking, and then the parser will try to match the input string with the next pattern in it, and if a pattern is matched by the input string, then there is no need to continue to match. Basing on such recognition, we use $<++$ in these functions. As for the function **munch**, we use it in detecting the comment with a lambda func-

tion in **munch** to judge if the current character is the end of the line, after matching the character , it will "eat" the characters until the end of the line, and combine the **ReadP** of these former successfully matched characters to a string's **ReadP** (and the string is the content of the comment). Because of this feature of this biased combinator, we use the function **munch** in detecting the comment and return the **ReadP String** that contains the content of the comment.

1.2 Some helper functions

Besides the skeleton mentioned above, our program also has a series of Helper functions (Auxiliary functions) that will be called in the skeleton like using function **intermediateOperExpHelper** and **innerOperExpHelper** to deal with left recursion, function **token** to skip the spaces around the string or character we want to match, function **extractIdent** to get the string in **Var**, function **rpComment** to identify the comments which will be used to clean the comments between the **Stmt**, and a series of **ReadP** of some characters like alphabets, digits, space and so on.

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
main function	parseString	Completed
Program	rpProgram	Completed
Stmts	rpStmts	Completed
Stmt	rpStmt	Completed
Expr	operExp	Completed
ident	rpIdent	Completed
numConst	concreteOperExp	Completed
stringConst	concreteOperExp	Completed
Auxiliary functions	token	Completed
Auxiliary functions	extractIdent	Completed
Auxiliary functions	rpComment	Completed

2.2 Correctness

After running on the online TA, we have 1 error and 7 timeout, remaining are OK. As for the time out cases, they are mostly about the deep parentheses, deep brackets, etc. Although we try to accelerate the matching speed by adding code like **do char '['; char '[' ; operExp; char ']'; char ']'** in functions, it doesn't work, maybe it is because we don't know what the testing case be like, and how deep the parentheses, brackets will be. And about errors, it is the case **not needed**. in fact, we know how to fix it, but we need much more time to debug (you know time is limited in this assignment), so we have no choice but to let it go. As for the error, the specific reason for the error is that our program specifies that the keyword must be separated by a space, e.g. **"notx"** would not be recognized as **Not (Var "x")** but as **Var "notx"** as a **Var**. however, it happens when the keywords immediately followed by parentheses(**"()"** or **"[]"**). OnlineTA tells us that **"not(x)"** should be correctly recognized as **Not (Var "x")**, but since our program need a space after the keyword, such a situation is not allowed, and the program will return an error. And the condition of correctness is as follows:

Class of Function	Function Name	Test Result
main function	parseString	OK
Program	rpProgram	OK
Stmts	rpStmts	OK
Stmt	rpStmt	OK
Expr	operExp	1 error; 7 time out; remains OK
ident	rpIdent	OK
numConst	concreteOperExp	OK
stringConst	concreteOperExp	OK
Auxiliary functions	token	OK
Auxiliary functions	extractIdent	OK
Auxiliary functions	rpComment	OK

2.3 Efficiency

The efficiency of our program might not so good this time because we have several time out testing cases on online TA involving deep parentheses, deep brackets, etc. Although we take some measures (as mentioned above) to reduce time complexity, it doesn't work, for we cannot fit any cases of deep parentheses, deep brackets, etc. in our code. Anyway, our code has high

efficiency in our test.

2.4 Robustness

In general, our program can identify illegal characters, illegal numbers, illegal variable names, and syntax errors. This is sufficient for most use cases, but we can't rule out some bugs that didn't take into account. The evaluation of function Robustness is as follows:

Class of Function	Function Name	Robustness
main function	parseString	Strong
Program	rpProgram	Strong
Stmts	rpStmts	Strong
Stmt	rpStmt	Strong
Expr	operExp	Strong
ident	rpIdent	Strong
numConst	concreteOperExp	Strong
stringConst	concreteOperExp	Strong
Auxiliary functions	token	Strong
Auxiliary functions	extractIdent	Strong
Auxiliary functions	rpComment	Strong

2.5 Maintainability

As for the maintainability, this time, there are a little code duplication in the form of copy-pasted segments with minor changes just for matching several input strings for parsing (especially the test cases on the online TA). So the maintainability of our code this time might not that good.

Class of Function	Function Name	Maintainability
main function	parseString	Good
Program	rpProgram	Good
Stmts	rpStmts	Good
Stmt	rpStmt	Good
Expr	operExp	not bad
ident	rpIdent	Good
numConst	concreteOperExp	Good
stringConst	concreteOperExp	Good
Auxiliary functions	token	Good
Auxiliary functions	extractIdent	Good
Auxiliary functions	rpComment	Good

A Appendix: BoaParser.hs

```haskell
-- Skeleton file for Boa Parser.

module BoaParser (ParseError, parseString) where

import BoaAST
import Data.Char
import Control.Applicative
import Text.ParserCombinators.ReadP
-- add any other other imports you need

type ParseError = String -- you may replace this

parseString :: String -> Either ParseError Program
parseString str = case readP_to_S rpProgram str of
  [] -> Left "Parsing Error"
  _  -> case snd (last (readP_to_S rpProgram str)) of
    "" -> Right (fst (last (readP_to_S rpProgram str)))
    _  -> Left "Invalid Input"



reserved :: [String]
reserved = ["None", "True", "False", "for", "if", "in", "not"]

-- Main skeleton

rpProgram :: ReadP Program
rpProgram = rpStmts

-- Clean the spaces and comments between statements
rpStmts :: ReadP [Stmt]
rpStmts = (do
                Text.ParserCombinators.ReadP.many space
                Text.ParserCombinators.ReadP.many rpComment
                Text.ParserCombinators.ReadP.many space
                stm <- rpStmt
                Text.ParserCombinators.ReadP.many space
                Text.ParserCombinators.ReadP.many rpComment
```

```
                Text.ParserCombinators.ReadP.many space
                token (char ';')
                stms <- rpStmts
                return (stm:stms))
        <++ (do
                Text.ParserCombinators.ReadP.many space
                Text.ParserCombinators.ReadP.many rpComment
                Text.ParserCombinators.ReadP.many space
                stm <- rpStmt
                Text.ParserCombinators.ReadP.many space
                Text.ParserCombinators.ReadP.many rpComment
                Text.ParserCombinators.ReadP.many space
                return [stm])

rpStmt :: ReadP Stmt
rpStmt = (do
        ident <- token rpIdent
        token (char '=')
        opex <- token operExp
        return (SDef (extractIdent ident) opex))
        <++ (do
                opex <- token operExp
                return (SExp opex))


operExp :: ReadP Exp
operExp = outerOperExp

outerOperExp :: ReadP Exp
outerOperExp = (do
                token (string "not ")
                e <- operExp
                return (Not e))
        <++ (do
                token (string "not")
                token (char '(')
                e <- operExp
                token (char ')')
                return (Not e))
        <++ (do
```

```
79                  token (string "not")
80                  Text.ParserCombinators.ReadP.many1 rpComment
81                  e <- operExp
82                  return (Not e))
83          <++ (do
84                  e1 <- intermediateOperExp
85                  token (string "==")
86                  e2 <- intermediateOperExp
87                  return (Oper Eq e1 e2))
88          <++ (do
89                  e1 <- intermediateOperExp
90                  token (string "!=")
91                  e2 <- intermediateOperExp
92                  ;return (Not (Oper Eq e1 e2)))
93          <++ (do
94                  e1 <- intermediateOperExp
95                  token (char '<')
96                  e2 <- intermediateOperExp
97                  return (Oper Less e1 e2))
98          <++ (do
99                  e1 <- intermediateOperExp
100                 token (string "<=")
101                 e2 <- intermediateOperExp
102                 return (Not (Oper Greater e1 e2)))
103         <++ (do
104                 e1 <- intermediateOperExp
105                 token (char '>')
106                 e2 <- intermediateOperExp
107                 return (Oper Greater e1 e2))
108         <++ (do
109                 e1 <- intermediateOperExp
110                 token (string ">=")
111                 e2 <- intermediateOperExp
112                 return (Not (Oper Less e1 e2)))
113         <++ (do
114                 e1 <- intermediateOperExp
115                 many1 space -- this and next lines are not equal to token
    ↪  (string " in ")
116                 token (string "in ")
117                 e2 <- intermediateOperExp
```

```
                return (Oper In e1 e2))
        <++ (do
                e1 <- intermediateOperExp
                token (string "not ")
                token (string "in ")
                e2 <- intermediateOperExp
                return (Not (Oper In e1 e2)))
        <++ intermediateOperExp

intermediateOperExp :: ReadP Exp
intermediateOperExp = (do
                e1 <- innerOperExp
                intermediateOperExpHelper e1)
        <++ innerOperExp

-- Using helper funciton to realize left association
intermediateOperExpHelper :: Exp -> ReadP Exp
intermediateOperExpHelper e1 = ( do
                token (char '+')
                e2 <- innerOperExp
                intermediateOperExpHelper (Oper Plus e1 e2))
        <++ ( do
                token (char '-')
                e2 <- innerOperExp
                intermediateOperExpHelper (Oper Minus e1 e2))
        <++ return e1

innerOperExp :: ReadP Exp
innerOperExp = (do
                e1 <- concreteOperExp
                innerOperExpHelper e1)
        <++ concreteOperExp

-- Using helper funciton to realize left association
innerOperExpHelper :: Exp -> ReadP Exp
innerOperExpHelper e1 = ( do
                token (char '*')
                e2 <- concreteOperExp
                innerOperExpHelper (Oper Times e1 e2))
        <++ ( do
```

```
158                 token (string "//")
159                 e2 <- concreteOperExp
160                 innerOperExpHelper (Oper Div e1 e2))
161         <++ ( do
162                 token (char '%')
163                 e2 <- concreteOperExp
164                 innerOperExpHelper (Oper Mod e1 e2))
165         <++ return e1
166
167 concreteOperExp :: ReadP Exp
168 concreteOperExp = (do
169                 s <- skipSpaces *> subtractChar
170                 ss <- many1 digit <* skipSpaces;
171                 if head ss == '0' && length ss > 1
172                 then
173                         fail "leading zero"
174                 else
175                         return  (BoaAST.Const (IntVal (read (s:ss)
    ↪  ::Int))))
176         <++ (do
177                 s <- skipSpaces *> many1 digit <* skipSpaces;
178                 if head s == '0' && length s > 1
179                         then
180                                 fail "leading zero"
181                         else
182                                 return  (BoaAST.Const (IntVal (read s
    ↪  ::Int))))
183         <++ (do
184                 char '\''
185                 grossContent <- Text.ParserCombinators.ReadP.many (
186                         stringChar
187                         <|> (do
188                                 string "\\n"
189                                 return '\n')
190                         <|> (do
191                                 string "\\\n"
192                                 return '\0')
193                         <|> (do
194                                 string "\\\\"
195                                 return '\\')
```

```
                        <|> (do
                                string "\\'"
                                return '\''))
                char '\'';
                let purifiedContent = filter (`notElem` "\NUL")
↪   grossContent in
                        return (BoaAST.Const (StringVal
↪   purifiedContent)))
        <++ (do
                token (string "None")
                return (BoaAST.Const NoneVal))
        <++ (do
                token (string "True")
                return (BoaAST.Const TrueVal))
        <++ (do
                token (string "False")
                return (BoaAST.Const FalseVal))
        <++ (do
                fNameVar <- token rpIdent
                token (char '(')
                opez <- concreteOperExpz
                token (char ')')
                return (Call (extractIdent fNameVar) opez))
        <++ rpIdent
        <++ (do
                token (char '(')
                token (char '(')
                ope <- operExp
                token (char ')')
                token (char ')')
                return ope)
        <++ (do
                token (char '(')
                ope <- operExp
                token (char ')')
                return ope)
        <++ (do
                token (char '[')
                token (char '[')
                opez <- concreteOperExpz
```

```
                token (char ']')
                token (char ']')
                return (List opez))
        <++ (do
                token (char '[')
                opez <- concreteOperExpz
                token (char ']')
                return (List opez))
        <++ (do
                token (char '[')
                ope <- operExp
                many1 space
                token (string "for ")
                ident <- token rpIdent
                token (string "in ")
                exp <- operExp
                cz <- rpClausez
                token (char ']')
                return (Compr ope (CCFor (extractIdent ident) exp:cz)))


concreteOperExpz :: ReadP [Exp]
concreteOperExpz = concreteOperExps
        <++ (do return [])

concreteOperExps :: ReadP [Exp]
concreteOperExps = (do
                e <- operExp
                token (char ',')
                es <- concreteOperExps
                return (e:es))
        <++ (do
                e <- operExp
                return [e])

rpClausez :: ReadP [CClause]
rpClausez = (do
                many1 space
                token (string "for ")
                ident <- token rpIdent
```

```
                token (string "in ")
                exp <- operExp
                cs <- rpClausez
                return (CCFor (extractIdent ident) exp:cs))
        <++ (do
                token (string "if ")
                exp <- operExp
                cs <- rpClausez
                return (CCIf exp:cs))
        <++ (do return [])


rpIdent :: ReadP Exp
rpIdent = do
    c <- letter'
    cs <- letdigs ;
                if (c:cs) `notElem` reserved
            then
                return (Var (c:cs))
            else
                fail "Cannot use reserved words as an identifier"
    where letter' = letter <|> underScore ;
           letdigs = Text.ParserCombinators.ReadP.many (alphaNum <|>
↪   underScore)

-- Helper Functions

token :: ReadP a -> ReadP a
token t = skipSpaces *> t <* skipSpaces

extractIdent :: Exp -> String
extractIdent (Var str) = str
extractIdent _         = ""

rpComment :: ReadP String
rpComment = (do
        char '#'
        comment <- munch (/= '\n')
        char '\n'
        return comment)
```

```
313         <++ (do
314         char '#'
315         munch (/= '\n'))
316
317
318 space :: ReadP Char
319 space = satisfy isSpace
320 digit :: ReadP Char
321 digit = satisfy isDigit
322 letter :: ReadP Char
323 letter = satisfy isLetter
324 alphaNum :: ReadP Char
325 alphaNum = satisfy isAlphaNum
326 subtractChar :: ReadP Char
327 subtractChar = satisfy isSubtract
328 stringChar :: ReadP Char
329 stringChar = satisfy canPrintChar
330 underScore :: ReadP Char
331 underScore = satisfy isUnderScore
332 -- table :: ReadP Char
333 -- table = satisfy isTable
334
335
336
337 isSubtract :: Char -> Bool
338 isSubtract s = case s of
339   '-' -> True
340   _   -> False
341
342 isUnderScore :: Char -> Bool
343 isUnderScore u = case u of
344   '_' -> True
345   _   -> False
346
347 -- isTable :: Char -> Bool
348 -- isTable t = case t of
349 --   '\t' -> True
350 --   _    -> False
351
352 canPrintChar :: Char -> Bool
```

```
353 canPrintChar c = case c of
354         '\'' -> False
355         '#'  -> True
356         '\\' -> False
357         _    -> isPrint c
358
```

B Appendix: Test.hs

```
1  -- Rudimentary test suite. Feel free to replace anything.
2
3  import BoaAST
4  import BoaParser
5
6  import Test.Tasty
7  import Test.Tasty.HUnit
8
9  main :: IO ()
10 main = defaultMain $ localOption (mkTimeout 1000000) tests
11
12 tests = testGroup "Minimal tests" [
13   testCase "simple success" $
14     parseString "2 + two" @?=
15       Right [SExp (Oper Plus (Const (IntVal 2)) (Var "two"))],
16   testCase "simple failure" $
17     parseString "2!" @?=
18       Left "Invalid Input",
19   testCase "simple number" $
20     parseString "-10086" @?=
21       Right [SExp (Const (IntVal (-10086)))],
22   testCase "test define & ';'" $
23     parseString "x = 1; y = 2" @?=
24       Right [SDef "x" (Const (IntVal 1)),SDef "y" (Const (IntVal 2))],
25   testCase "test spaces" $
26     parseString "    1     +      2      *      3      " @?=
27       Right [SExp (Oper Plus (Const (IntVal 1)) (Oper Times (Const
↪  (IntVal 2)) (Const (IntVal 3))))],
```

```
28    testCase "test lists & String" $
29      parseString "[[1,2,3],['one','two','three']]" @?=
30        Right [SExp (List [List [Const (IntVal 1),Const (IntVal 2),Const
      ↪  (IntVal 3)],List [Const (StringVal "one"),Const (StringVal
      ↪  "two"),Const (StringVal "three")]])] ,
31    testCase "test comput1" $
32      parseString "-1+23-(-456)" @?=
33        Right [SExp (Oper Minus (Oper Plus (Const (IntVal (-1))) (Const
      ↪  (IntVal 23))) (Const (IntVal (-456))))],
34    testCase "test comput2" $
35      parseString "(1+2)-3*4>=5%6-7//8" @?=
36        Right [SExp (Not (Oper Less (Oper Minus (Oper Plus (Const (IntVal
      ↪  1)) (Const (IntVal 2))) (Oper Times (Const (IntVal 3)) (Const (IntVal
      ↪  4)))) (Oper Minus (Oper Mod (Const (IntVal 5)) (Const (IntVal 6)))
      ↪  (Oper Div (Const (IntVal 7)) (Const (IntVal 8))))))],
37    testCase "test comput3" $
38      parseString "(1*(2//(3%4)))" @?=
39        Right [SExp (Oper Times (Const (IntVal 1)) (Oper Div (Const (IntVal
      ↪  2)) (Oper Mod (Const (IntVal 3)) (Const (IntVal 4)))))],
40    testCase "test ccfor & ccif" $
41      parseString "[x for y in [1,2,3] if (y != 0)]" @?=
42        Right [SExp (Compr (Var "x") [CCFor "y" (List [Const (IntVal
      ↪  1),Const (IntVal 2),Const (IntVal 3)]),CCIf (Not (Oper Eq (Var "y")
      ↪  (Const (IntVal 0))))])],
43    testCase "test Eq" $
44      parseString "((1<2)==(4>=3))!=False" @?=
45        Right [SExp (Not (Oper Eq (Oper Eq (Oper Less (Const (IntVal 1))
      ↪  (Const (IntVal 2))) (Not (Oper Less (Const (IntVal 4)) (Const (IntVal
      ↪  3))))) (Const FalseVal)))],
46    testCase "test string" $
47      parseString "\n \t 1 \t + 1 \n; \n x \t = 1" @?=
48        Right [SExp (Oper Plus (Const (IntVal 1)) (Const (IntVal 1))),SDef
      ↪  "x" (Const (IntVal 1))],
49    testCase "test comment" $
50      parseString "# test 1 \n 1 # test 2 \n" @?=
51        Right [SExp (Const (IntVal 1))],
52    testCase "test syntax error" $
53      parseString "1(-1)" @?=
54        Left "Invalid Input",
55    testCase "test keyword error" $
```

```
56      parseString "for = 1" @?=
57        Left "Invalid Input",
58    testCase "test name def1" $
59      parseString "_t_est_ = 1" @?=
60        Right [SDef "_t_est_" (Const (IntVal 1))],
61    testCase "test name def2" $
62      parseString "0Test = 1" @?=
63        Left "Invalid Input",
64    testCase "test number error" $
65      parseString "-001" @?=
66        Left "Invalid Input",
67      case parseString "wow!" of
68        Left e -> return ()  -- any message is OK
69        Right p -> assertFailure $ "Unexpected parse: " ++ show p]
70
```