Home Assignment 3
Advanced Programming

hkg795, knv765

September 30, 2022

1 Design and Implementation

In order to assist our implementation and minimise logical errors, we decided to transform the given context free grammar into BNF grammar with eliminated left recursion respecting the precedence of calculative expressions. Since our Correctness assumption relies heavily on a correct grammar we go into further detail about this in 2.2.

Other than that we mostly based our implementation on what was presented in the lectures. Using parser combinators allows us to combine both lexing and syntactic analysis. This makes parsing a String almost as readable as the grammar itself. Thus the code should be mostly self explanatory, except for maybe the parseStringConst function where parsing special characters introduced some challenges with escaping.

The only instance where the <++ and munch operation where necessary was when parsing comments. This is because the second option (skipWS) should only be tried when there's no # to be found since both alternatives could consume (or not consume) some amount of Whitespace which would result in issues with the backtracking. Munch was used simply to consume the entire comment line.

We do not know why onlineTA keeps complaining about our package.yaml since it should be identical to the original one.

2 Assessment

2.1 Completeness

All the required functionality has been implemented according to the specifications and based on our modified grammar as listed in 3.1 using readP.

The optional Parsec warm up Task was not implemented.

2.2 Correctness

For the correctness of the assignment we relied heavily on a transformed BNF (3.1) of the original context free grammar for the Boa syntax. We have removed left recursion in the statements, relation, multiplication and division, addition and negation expressions, using the example from the slides. We made sure to keep multiplication, division, addition and negation left associative while respecting the precedence of the calculations. The correct precedence is respected through using the defined expressions in a way that a *RelExpr* can use only *AddNegExpr* which can use only *MultDivExpr*, and so on.

In our current implementation we are handling expressions involving brackets inefficiently. Whilst the code does theoretically work, onlineTA's tests regarding deep brackets fail because of a time out. This issue is discussed on the Efficiency part of the report.

In order to make sure we handle general and edge cases correctly, we have relied on bottom up adhoc testing during the lexing of individual tokens and on top down structured tests to track down logic errors once the lexing parsers implementation was completed. Testing has been done through the provided test suite and through the OnlineTA insights. The test suite is divided into parts corresponding to descriptions in the assignment and grouped into

- Identifier
- Numerical Constants
- String Constants
- Whitespace around tokens
- Comments
- Disambiguation
- Concrete and abstract syntax correspondence
- General

We have found an issue during testing and after passing onlineTA but unfortunately too late to investigate it further. It is described in the "Illegal new line" test case. We expected an error but incorrectly parsed.

2.3 Efficiency

We believe our code to be an adequately efficient implementation of a Boa parser in Haskell according to the provided specifications with the exception of deep brackets

Both the expressions of List and List Comprehension, as described in our grammar, start by parsing a ' [' token. When parsing an input that involves a heavily nested List or a List Comprehension, our parser takes a long time to go through. This problem occurs as the current form of our grammar does not help the parser make a quick decision of which expression to go through. This introduces deep backtracking and longer than usual computation times. This issue could be fixed by reforming the List and List Comprehension expressions or perhaps through a biased parser choice through the (< ++) operator. The brackets issue does not occur with deep parentheses, as there is essentially only way to parse them; through nesting one or more *Expr*.

2.4 Robustness

Every Haskell-type correct input (that is, calling parseString and passing a String) should result in a well formed output.

2.5 Maintainability

We believe our code to be in a well maintainable form, due to extensive comments regarding implementation details and in most cases a direct translation of the specifications into the functional language. Common code snippets were extracted into helper functions and the code follows a consistent layout.

3 Appendix

3.1 BNF Grammar

```
Program ::= Stmts
Stmts ::= Stmt Stmts'
Stmts' ::= ';' Stmts | empty
Stmt ::= ident '=' Expr | Expr

Expr ::=  'not' Expr | RelExpr
RelExpr ::= AddNegExpr RelExpr'
RelExpr' ::= RelOper AddNegExpr | empty
```

```
AddNegExpr ::= MultDivExpr AddNegExpr'
AddNegExpr' ::= AddNegOper MultDivExpr AddNegExpr' | empty
MultDivExpr ::= ConstExpr MultDivExpr'
MultDivExpr' ::= MultDivOper ConstExpr MultDivExpr' | empty
ConstExpr ::= stringConst | numConst
| 'None' | 'True' | 'False'
| ident | '(' Expr ')'
| ident '(' Exprz ')'
| '[' Expr ForClause Clausez ']'
| '[' Exprz ']'

MultDivOper ::= '*' | '//' | '%'
AddNegOper ::= '+' | '-'
RelOper ::= '==' | '!=' | '<' | '<='
| '>' | '>=' | 'in' | 'not' 'in'
ForClause ::= 'for' ident 'in' Expr
IfClause ::= 'if' Expr
Clausez ::= empty
| ForClause Clausez
| IfClause Clausez
Exprz ::= empty
| Exprs
Exprs ::= Expr Exprs'
Exprs' ::= ',' Exprs | empty
numConst ::= (1-9)(0-9)* | 0 | '-'0 | '-'(1-9)(0-9)*
ident ::= (followed assignment)
stringConst ::= (followed assignment)
```

3.2 Code

3.2.1 BoaParser.hs

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

module BoaParser (ParseError, parseString) where

import BoaAST
-- add any other other imports you need

import Text.ParserCombinators.ReadP
import Control.Applicative ((<|>)) -- may use instead of +++
  --for easier portability to Parsec
```

```
import Data.Char (isDigit, isSpace, isLetter, isPrint)

type Parser a = ReadP a   -- may use synomym
    --for easier portability to Parsec
type ParseError = String -- you may replace this

reservedIdents = ["None", "True", "False", "for", "if", "in", "not"]

--parses a boa program according to
--the modified grammar in BNF_grammar.txt
--using parser combinators allows us to
--combine both lexical and syntactical analysis
parseString :: String -> Either ParseError Program
parseString s  = case [a | (a,t) <- readP_to_S parseProgram s,
    all isSpace t] of
              [a] -> Right a
              [] -> Left "Parsing failed"
              _ -> Left "How did it get here ?"

parseProgram :: Parser Program
parseProgram = do
                skipWS
                parseStmts

--skipWS (skips whitespace and comments)
--is used overly cautious all throughout our code
--many of these are superflous since every parser should
--already skip all the WS after it's token
--(and parseProgramm at the very beginning)
--nonetheless, this ensures a correct parse
--and is only a minor drawback with regards to efficiency
parseStmts :: Parser [Stmt]
parseStmts = do
                stmt <- parseStmt
                skipWS --redundant
                rest <- parseStmts'
                skipWS
                return (stmt:rest)

--epsilon production handled last
parseStmts' :: Parser [Stmt]
```

```
parseStmts' = do
    satisfy (== ';')
    skipWS
    parseStmts
    <|> do
    return []

parseStmt :: Parser Stmt
parseStmt = do
            ident <- parseIdent
            skipWS
            satisfy(== '=')
            skipWS
            expr <- parseExpr
            skipWS
            return $ SDef ident expr
            <|> do
            expr <- parseExpr
            skipWS
            return $ SExp expr


parseExpr :: Parser Exp
parseExpr = do
    parseKeyWord "not"
    skipWS
    exp <- parseExpr
    skipWS
    return $ Not exp
    <|> do
    parseRel

--this function parses keywords like not, in etc...
--we use look to peak at the letter following the
--keyword without consuming it.
--if this letter could belong to an identifier
--then we can not parse a keyword like for ex. in "notx"
--this is necessary bc "not(x)" should parse to an Expr
parseKeyWord :: String -> Parser String
parseKeyWord str = do
    string str
```

```
    next <- look
    skipWS
    if (\x -> not (isDigit x || isLetter x || x == '_')) $ head next
        then return str else pfail

parseRel :: Parser Exp
parseRel = do
              exp <- parseAddNeg
              skipWS
              parseRel' exp

parseRel' :: Exp -> Parser Exp
parseRel' expr = do
                   parseRelOper expr
                   <|>
                   return expr

parseRelOper :: Exp -> Parser Exp
parseRelOper expr1 = do
                   string "=="
                   skipWS
                   expr2 <- parseAddNeg
                   return $ Oper Eq expr1 expr2
                   <|> do
                   string "!="
                   skipWS
                   expr2 <- parseAddNeg
                   return $ Not $ Oper Eq expr1 expr2
                   <|> do
                   satisfy (== '<')
                   skipWS
                   expr2 <- parseAddNeg
                   return $ Oper Less expr1 expr2
                   <|> do
                   satisfy (== '>')
                   skipWS
                   expr2 <- parseAddNeg
                   return $ Oper Greater expr1 expr2
                   <|> do
                   string "<="
                   skipWS
```

```
                    expr2 <- parseAddNeg
                    return $ Not $ Oper Greater expr1 expr2
                    <|> do
                    string ">="
                    skipWS
                    expr2 <- parseAddNeg
                    return $ Not $ Oper Less expr1 expr2
                    <|> do
                    parseKeyWord "in"
                    expr2 <- parseAddNeg
                    return $ Oper In expr1 expr2
                    <|> do
                    parseKeyWord "not"
                    skipWS
                    parseKeyWord "in"
                    expr2 <- parseAddNeg
                    return $ Not $ Oper In expr1 expr2

parseAddNeg :: Parser Exp
parseAddNeg = do
              m <- parseMultDiv
              skipWS
              parseAddNeg' m

parseAddNeg' :: Exp -> Parser Exp
parseAddNeg' expr = do
                       satisfy(== '+')
                       skipWS
                       m <- parseMultDiv
                       skipWS
                       parseAddNeg' $ Oper Plus expr m
                       <|> do
                       satisfy(== '-')
                       skipWS
                       m <- parseMultDiv
                       skipWS
                       parseAddNeg' $ Oper Minus expr m
                       <|>
                       return expr

parseMultDiv :: Parser Exp
```

```haskell
parseMultDiv = do
             m <- parseConst
             skipWS
             parseMultDiv' m

parseMultDiv' :: Exp -> Parser Exp
parseMultDiv' expr = do
                      satisfy(== '*')
                      skipWS
                      m <- parseConst
                      skipWS
                      parseMultDiv' $ Oper Times expr m
                      <|> do
                      string "//"
                      skipWS
                      m <- parseConst
                      skipWS
                      parseMultDiv' $ Oper Div expr m
                      <|> do
                      satisfy(=='%')
                      skipWS
                      m <- parseConst
                      skipWS
                      parseMultDiv' $ Oper Mod expr m
                      <|>
                      return expr

--the last to productions in this expression
--namely List and List Comprehnsion expressions
--cause some major efficieny issues when
--parsing deep brackets. this is because both
--productions start with the same symbol
--and due to the nature of built in backtracking
--in readP we have to reevaluate (parts) of the
--input several times skipping back and forth
--a remedy might exist using the <++ operator
parseConst :: Parser Exp
parseConst = do
    parseStringConst
    <|> do
    parseNumConst
```

```
<|> do
ident <- parseIdent
skipWS
return $ Var ident
<|> do
string "None"
skipWS
return $ Const NoneVal
<|> do
string "True"
skipWS
return $ Const TrueVal
<|> do
string "False"
skipWS
return $ Const FalseVal
<|> do
satisfy (== '(')
skipWS
exp <- parseExpr
skipWS
satisfy (== ')')
skipWS
return exp
<|> do --fun call syntax
fname <- parseIdent
skipWS
satisfy (== '(')
skipWS
args <- parseExprz
skipWS
satisfy (== ')')
skipWS
return $ Call fname args
<|> do --eval list syntax
satisfy (== '[')
skipWS
exprz <- parseExprz
skipWS
satisfy (== ']')
skipSpaces
```

```
    return $ List exprz
    <|> do --list comp syntax
    satisfy (== '[')
    skipWS
    exp <- parseExpr
    skipWS
    for <- parseForClause
    skipWS
    rest <- parseClausez
    skipWS
    satisfy (== ']')
    skipSpaces
    return $ Compr exp (for:rest)

parseForClause :: Parser CClause
parseForClause = do
    parseKeyWord "for"
    ident <- parseIdent
    skipWS
    parseKeyWord "in"
    exp <- parseExpr
    skipWS
    return $ CCFor ident exp

parseIfClause :: Parser CClause
parseIfClause = do
    parseKeyWord "if"
    exp <- parseExpr
    return $ CCIf exp

parseClausez :: Parser [CClause]
parseClausez = do
    for <- parseForClause
    rest <- parseClausez
    return (for:rest)
    <|> do
    iff <- parseIfClause
    rest <- parseClausez
    return (iff:rest)
    <|> return []
```

```
parseExprz :: Parser [Exp]
parseExprz = do parseExprs;
             <|> return []

parseExprs :: Parser [Exp]
parseExprs = do
    exp <- parseExpr
    skipWS
    rest <- parseExprs'
    return (exp:rest)

parseExprs' :: Parser [Exp]
parseExprs' = do
    satisfy (== ',')
    skipWS
    parseExprs
    <|> return []

--just skipSpaces should only be applied
--if we have no comments to skip
--hence, the <++ operator is used to
--only check this option if no # could be found
skipWS :: Parser ()
skipWS = do
    skipSpaces
    satisfy (== '#')
    skipComments
    <++ do  --doc this
    skipSpaces

skipComments :: Parser ()
skipComments = do
    munch (/= '\n')
    skipCommentsEnd

--a comment either ends with \n or
--at the end of the input
skipCommentsEnd :: Parser ()
skipCommentsEnd = do
    eof
    <|> do
```

```haskell
    string "\n"
    skipWS
    return mempty

--the following functions parse the complex terminals
--according to the specifications
parseIdent :: Parser String
parseIdent = do
    ident <- munch1 (\x -> isDigit x || isLetter x || x == '_')
    skipWS
    if isDigit (head ident) || ident `elem` reservedIdents
        then pfail else return ident

parseNumConst :: Parser Exp
parseNumConst = do
    satisfy (== '-')
    num <- parseNumConstHelper
    return $ Const (IntVal (-num))
    <|> do
    num <- parseNumConstHelper
    return $ Const (IntVal num)

parseNumConstHelper :: Parser Int
parseNumConstHelper = do
    num <- munch1 isDigit
    skipWS
    case head num of
        '0' -> if length num == 1 then return 0 else pfail
        _  -> return $ read num

parseStringConst :: Parser Exp
parseStringConst = do
    satisfy (== '\'')
    print <- many parseStringInside
    satisfy (== '\'')
    skipWS
    return $ Const (StringVal $ concat print)

parseStringInside :: Parser String
parseStringInside = do
                    c <- satisfy isPrintable
```

```
                    return [c]
                    <|> do
                    string "\\\n"
                    return ""
                    <|> do
                    string "\\n"
                    return "\n"
                    <|> do
                    string "\\\'"
                    return "\'"
                    <|> do
                    string "\\\\"
                    return "\\"

isPrintable :: Char -> Bool
isPrintable c = isPrint c && (c /= '\'') && (c /= '\\')
```