**Advanced Programming - Fall 2019**

Assignment 3

*Till Grenzdörffer, Michal Pikulski*

Handed in: September 25, 2019

# Contents



## Design considerations and encountered problems

- We decided to use the Parsec library primarily for the detailed error messages, which turned out to help us a lot with debugging our code.
- In order to deepen our understanding of the monadic structure of Parsec, we decided to avoid using do notation for the parser. It turns out that functions look more compact and readable, once you get used to it. Also, we tried using Functor and Applicative operations wherever possible.
- As a general policy, we used the try keyword only, if there are two or more alternatives that could possibly start with the same symbol. Notable uses of backtracking can be found in:
    - Relational operators - because they often share a prefix
    - Boolean Values - because identifiers could contain them as a prefix
    - In `relExpr'` - it's enforced by the way our grammar is written. To reflect relational operators being non associative, we do not allow the original production `relExpr` to evaluate to an additive expression. Because of that, we need to back track in case we actually are evaluating a standalone `addExpr`.
    - In `statements` - unfortunately, we have an indirect ambiguity here. A statement can start with an identifier or can evaluate to an expression. Said expression, however, can also evaluate to an identifier. We did not find a graceful way of disambiguating grammar, so we chose to use backtracking.
- We treat comments as whitespaces, so our `whitespace` parser skips over them.

- We wrote multiple helper functions for different kinds of terminals. Some don't need whitespace following them, like '=' and other arithmetic symbols, some need not to be followed by anything alphaNumeric, some need at least some trailing whitespace.
    - Said helper functions could be done better, possibly merged into one generic function (we have 3 different keywords now!) that is parametrized by the type of keyword we are expecting. We tried looking into that, but since Parsec does not allow alternatives in `manyOf` that evaluate to different kinds of parsers, we left those functions the way they are.
- As proposed in the lecture, every terminal gets rid of trailing whitespace using the lexeme paradigm. Additionally, any potential whitespaces/comments at the beginning of the program are handled separately in the "program" production.
- In the `stringConst` function we need to be able to return an empty string, if we parse \n. Since `char` can not be empty, we parse the escaping symbols as strings. Since we have many of them, we have a parser of lists of strings, which we need to concat afterwards. That seems not very ideal, but we did not find another solution.
- For the operator precedence and associativity we rewrote the grammar. The resulting grammar can be found in the appendix. First we rewrote it to account for precedence, then to account for associativity and only then we removed left-recursion. We found that to be the least confusing way of doing it.
- Additionally, we removed some ambiguous productions, limiting the amount of backtracking we were later required to do, using Parsec's "try" function.
- In our implementation, we consider an empty string to not be a program. We did not find any value in allowing that, and the implementation would have to be even more confusing and account for even more ambiguity.

# Assessment of code quality

## Quality assessment

Since we wrote a property test that spans the majority of the parser's functionality, we are fairly confident that we found the majority of bugs in the parser. Given the time we think we found a good solution to the problem. However, we know that there are several parts of our code that could be cleaned up for better readability and maintainability:

- Keywords - we parse certain keywords differently than others, as some require spaces after them, can be included in identifier names etc. We think that, given time, we could rework those functions into a better, more readable solution
- Backtracking - while in some instances necessary, there is a particular place where we could have possibly modified the grammar some more and therefore avoid using backtracking
- In tests, we could have spent a little bit more time on our arbitrary data type, to ensure it only generates values that we are able to parse. There were some corner cases (namely empty lists) that we had to manually account for.

Otherwise than that, we are certain that our code is at the very least adequate and solves the problem in a correct and bug-free way.

## Testing strategy

As proposed in one of the lectures, we wrote a pretty-printer for boa-ASTs. We also defined a full arbitrary data structure that allows QuickCheck to automatically generate example elements of our AST. Having that, we decided to test two things - expressions and full programs. We considered testing all the separate token functions, but given that all of them can be included in a program, we decided against it. To ensure proper coverage we have locally run our test for large inputs (over 10000 generated programs per run) and given the recursive nature of our data structures we are certain it covers enough of the possible variations, that writing unit tests for those functions feels unnecessary. In the submitted package, though, the number of tests has been cut down to limit the amount of time they take to execute.

There are some assumptions/constraints that exist in our test:

- We limit any generated strings to 25 characters. It bears no influence on test results (to confirm that, we did run the tests on unlimited strings), but cuts down execution time drastically.
- For any input n we limit the depth of our structure, so the running times remain within reason. That's why we use `"div 4"` and not `"div 2"` when traversing down the structures.
- We did not account for the arbitrary data type generating some inputs that we are unable to parse. For those inputs we have separate tests, which directly test the expected output of such corner cases.

Given the amount of tests and the repeated local execution, we are fairly certain that the entirety of our program is well covered. As mentioned in the report - the submitted assignment has a significantly reduced repetitions, given the amount of time it would take to process large inputs.

# Appendix 1 - Modified Grammar

```
Program ::= Stmts

Stmts ::= Stmt
       | Stmt ';' Stmts

Stmt ::= ident '=' Expr
       | Expr

Expr ::= RelExpr | 'not' Expr

RelExpr ::= AddExpr | AddExpr RelOp AddExpr

AddExpr  ::= MulExpr | MulExpr AddExpr'
AddExpr' ::= e | AddOp MulExpr AddExpr'

MulExpr ::= Term | Term MulExpr'
MulExpr' ::= e | MulOp Term MulExpr'

Term ::= numConst
       | stringConst
       | 'None' | 'True' | 'False'
       | ident IdentFun
       | '(' Expr ')'
       | '[' Exprz ']'
       | '[' Expr ForQual Qualz ']'


IdentFun ::= e | '(' Exprz ')'

Oper ::= AddOp
       | MultOp
       | RelOp

AddOp   ::= '+' | '-'
MultOp  ::= '*' | '//' | '%'
RelOp   ::= '==' | '!=' | '<' | '<=' | '>' | '>=' | 'in' | 'not' 'in'

Exprz ::= e
       | Exprs

Exprs ::= Expr
       | Expr ',' Exprs

ident ::= (see text)
numConst ::= (see text)
stringConst ::= (see text)
```

# Appendix 2 - code listing

## BoaParser.hs

```
-- Skeleton file for Boa Parser.

module BoaParser (ParseError, parseString) where

import BoaAST
import Text.ParserCombinators.Parsec
import Data.Char
import Control.Applicative (liftA2)

ident :: Parser String
ident = lexeme $ ((liftA2 (:) lu (many ldu)) >>= (\id -> if (id `elem` reservedKeywords) then fail
(id ++ " is a reserved keyword.") else return id))

numConst :: Parser Int
numConst = lexeme $ (read <$> (:[]) <$> (char '0') -- abuse no backtracking! 3rd case will not be
used if leading 0 present
                <|>
                ((char '-') >> (negate <$> numConst))
                <|>
                read <$> many1 digit)

stringConst :: Parser String
stringConst = lexeme $ (char '\'') *> (printEscaped) <* (char '\'')

listComp :: Parser Exp
listComp = liftA2 Compr expr (liftA2 (:) (forQual) (many $ (forQual <|> ifQual)))

ifQual :: Parser Qual
ifQual = (singletonKeyword "if") *> (QIf <$> expr)

forQual :: Parser Qual
forQual = (singletonKeyword "for") *> liftA2 QFor ident (singletonKeyword "in" *> expr)

parseString :: String -> Either ParseError Program
parseString s = (parse program "BoaParser" s)

program :: Parser Program
program = whitespace >> (stmt `sepBy1` (symbol ';')) <* eof

stmt :: Parser Stmt
stmt =  (try (liftA2 SDef ident (symbol '=' >> expr)))
        <|>
        (try (SExp <$> expr))

expr :: Parser Exp
expr  = (try (singletonKeyword "not") *> (Not <$> expr))
        <|>
        relExpr

symbol :: Char -> Parser Char
symbol k = (lexeme $ (char k))

relExpr :: Parser Exp
relExpr = (addExpr >>= relExpr')
```

```
relExpr' :: Exp -> Parser Exp
relExpr' e1 = try (relOp >>= (\oper -> ((oper e1) <$> addExpr) >>= addExpr'))
        <|> return e1

addExpr :: Parser Exp
addExpr = (multExpr >>= addExpr')
        <|> multExpr

addExpr' :: Exp -> Parser Exp
addExpr' e1 = (addOp >>= (\oper -> ((oper e1) <$> multExpr) >>= addExpr'))
        <|> return e1


multExpr :: Parser Exp
multExpr = (term >>= multExpr')

multExpr' :: Exp -> Parser Exp
multExpr' e1 = (multOp >>= (\oper -> ((oper e1) <$> term) >>= multExpr'))
        <|> return e1

relOp :: Parser (Exp -> Exp -> Exp)
relOp = lexeme $ (keyword "==" *> (return $ Oper Eq)
                <|>
                keyword "!=" *> (return $ (\a b -> Not (Oper Eq a b)))
                <|>
                try( keyword ">=" *> (return $ (\a b -> Not (Oper Less a b))))
                <|>
                try( keyword "<=" *> (return $ (\a b -> Not (Oper Greater a b))))
                <|>
                keyword ">" *>  (return $ Oper Greater)
                <|>
                keyword "<" *>  (return $ Oper Less)
                <|>
                (string "in" <* notFollowedBy alphaNum <* whitespace) *> (return $ Oper In)
                <|>
                (try (keyword1 "not" *> keyword "in" *> (return $ (\a b -> Not (Oper In a b))))))

addOp :: Parser (Exp -> Exp -> Exp)
addOp = lexeme $ ( symbol '+' *> (return $ Oper Plus)
                <|>
                symbol '-' *> (return $ Oper Minus))

multOp :: Parser (Exp -> Exp -> Exp)
multOp = lexeme $ ( symbol '*' *> (return $ Oper Times)
                <|>
                string "//" *> (return $ Oper Div)
                <|>
                symbol '%' *> (return $ Oper Mod))

term :: Parser Exp
term = (try $ List <$> (brackets exprz))
        <|>
        (brackets listComp)
        <|>
        (Const <$> StringVal <$> stringConst)
        <|>
        (Const <$> IntVal <$> numConst)
        <|>
        (parens expr)
        <|>
```

```
        (try (singletonKeyword "None" *> return (Const NoneVal)))
        <|>
        (try (singletonKeyword "False" *> return (Const FalseVal)))
        <|>
        (try (singletonKeyword "True" *> return (Const TrueVal)))
        <|>
        (ident >>= identFun)

identFun :: String -> Parser Exp
identFun e1 = (Call e1 <$> (parens exprz))
               <|>
               (return $ Var e1)

exprz :: Parser [Exp]
exprz = expr `sepBy` (symbol ',')

parens = between (symbol '(') (symbol ')')
brackets = between (symbol '[') (symbol ']')

-- Helper functions for handling whitspaces
whitespace :: Parser ()
whitespace = skipMany ( ((satisfy isSpace) *> return "") <|> (((char '#') *> skipMany (noneOf "\n")
*> (skipMany1 newline <|> eof) *> return " ")))

whitespace1 :: Parser ()
whitespace1 = skipMany1 (((satisfy isSpace) *> return "") <|> (((char '#') *> skipMany (noneOf "\n")
*> (skipMany1 newline <|> eof) *> return " ")))

lexeme :: Parser a -> Parser a
lexeme p = p <* whitespace

lexeme1 :: Parser a -> Parser a
lexeme1 p = p <* whitespace1

toString :: Parser Char -> Parser String
toString c = (:[]) <$> c

-- Handle different types of keywords
lu = ((satisfy isAlpha) <|> char '_')
ldu = (alphaNum <|> char '_')

keyword :: String -> Parser String
keyword k = (lexeme $ (string k))

keyword1 :: String -> Parser String
keyword1 k = (lexeme1 $ (string k))

singletonKeyword :: String -> Parser String
singletonKeyword s = (string s <* (notFollowedBy alphaNum) <* whitespace)

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

--  printable chars without \ or '
printableNoQBS :: Char -> Bool
printableNoQBS c = (isPrint c) && (not $ c `elem` ['\\', '\''])

-- parse everything as string, then concat. Because we need many 'something' where 'something' can
be the
-- empty string, thus we must use many over strings...
printEscaped :: Parser String
printEscaped = concat <$> (many $ ((toString (satisfy printableNoQBS))
```

```
<|>
(try ((char '\\') *> (string "'")))  -- \'
<|>
(try ((char '\\') *> (string "\\")))  -- \\
<|>
(try ((char '\\' *> (char 'n') *> return "\n")))
<|>
(((char '\\') *> (char '\n') *> return "")))
```

## Test.hs

```
-- Rudimentary test suite. Feel free to replace anything.

import BoaAST
import BoaParser

import Control.Monad
import Test.Tasty
import Test.Tasty.HUnit
import qualified Test.Tasty.QuickCheck as QC
import Data.Char
import Control.Applicative (liftA2)
import Text.Parsec.Error

main :: IO ()
main = defaultMain $ localOption (mkTimeout 10000000) tests

tests :: TestTree
tests = testGroup "All tests" [minimalTests, propertyTst, cornerCases]

isError :: String -> Bool
isError s = case parseString s of
                (Left err) -> True
                (Right _)  -> False

cornerCases :: TestTree
cornerCases = testGroup "Testing some corner cases"
        [testCase "Empty list"
        (assertBool "" ((Right [SExp (List [])]) == parseString("[]"))),
        testCase "Empty program"
        (assertBool "" (True == (isError "")))]

propertyTst :: TestTree
propertyTst = testGroup "Property tests" [prop_encode_decode]

prop_encode_decode = testGroup "Test encode-decode"
                [ QC.testProperty "Test encode-decode on expression" $ (QC.withMaxSuccess 200
                (\s -> let string = printStatement s in
                                case parseString string of
                                (Right (stmt:[])) ->  stmt QC.=== s
                                a -> (Right []) QC.=== a
                )),
                QC.testProperty "Test encode-decode on whole program" $ (QC.withMaxSuccess 50
                (\p -> let string = printProgram p in
                                case string of
                                "[]" -> 1 QC.=== 1
                                "" -> 1 QC.=== 1
                                s -> case parseString s of
                                        (Right stmts) ->  stmts QC.=== p
                                        a -> (Right []) QC.=== a
                ))
                ]

minimalTests :: TestTree
minimalTests = testGroup "Minimal tests" [
  testCase "simple success" $
        parseString "2 + two" @?=
        Right [SExp (Oper Plus (Const (IntVal 2)) (Var "two"))],
```

```
  testCase "simple failure" $
        -- avoid "expecting" very specific parse-error messages
        case parseString "wow!" of
        Left _ -> return ()  -- any message is OK
        Right p -> assertFailure $ "Unexpected parse: " ++ show p]

-- identifier
newtype Identifier = Ident String deriving (Eq, Show)
instance QC.Arbitrary Identifier where
  arbitrary = Ident <$> (take 25) <$> (liftA2 (:) (QC.arbitrary `QC.suchThat` lu) (QC.listOf
(QC.arbitrary `QC.suchThat` ldu))) `QC.suchThat` notKeyword

lu c = isAlpha c || (c == '_')
ldu c = (isAlpha c || (c == '_'))
printableNoQBS c = (isPrint c) && (not $ c `elem` ['\\', '\''])
notKeyword s = not $ s `elem` reservedKeywords
reservedKeywords = ["None", "True", "False", "for", "if", "in", "not"]

newtype StringConst = SC String deriving (Eq, Show)
instance QC.Arbitrary StringConst where
  arbitrary = SC <$> concat <$> QC.listOf validTokens

validTokens = liftA2 (:) ((QC.arbitrary `QC.suchThat` printableNoQBS)) (QC.elements
(["\\\\","\\\'","\\n","\\\n"]))

-- value
instance QC.Arbitrary Value where
  arbitrary = val

-- we don't need listvalues since everything is parsed as List [exp] anyways
val =  QC.oneof $ [ return NoneVal,
                return TrueVal,
                return FalseVal,
                liftM IntVal QC.arbitrary,
                do (SC s) <- QC.arbitrary
                return $ StringVal (take 25 s)
        ]

-- expressions
instance QC.Arbitrary Exp where
  arbitrary = sizedExp

sizedExp = QC.sized expN

expN 0 = QC.oneof $ [ liftM Const QC.arbitrary,
                        (do (Ident name) <- QC.arbitrary
                        return $ Var name),
                        liftM Not QC.arbitrary,
                        return (List []),
                        (do o <- QC.arbitrary
                        v1 <- QC.arbitrary
                        v2 <- QC.arbitrary
                        return $ Oper o (Const v1) (Const v2)),
                        (do (Ident name) <- QC.arbitrary
                        v <- QC.arbitrary
                        return $ Call name [(Const v)]),
                        (do v1 <- QC.arbitrary
                        v2 <- QC.arbitrary
                        (Ident name) <- QC.arbitrary
                        return $ Compr (Const v1) [(QFor name (Const v2))])]
```

```
expN n = QC.oneof $ [ liftM Const QC.arbitrary,
                      (do (Ident name) <- QC.arbitrary
                      return $ Var name),
                      liftM Not QC.arbitrary,
                      return (List []),
                      (do o <- QC.arbitrary
                      e1 <- subExp
                      e2 <- subExp
                      return $ Oper o e1 e2),
                      (do (Ident name) <- QC.arbitrary
                      rest <- QC.arbitrary
                      return $ Call name [rest]),
                      (do e1 <- subExp
                      e2 <- subExp
                      (Ident name) <- QC.arbitrary
                      (Quals quals) <- subQual
                      return $ Compr e1 ((QFor name e2):quals))]
                      where subExp = expN (n `div` 4)
                      subQual = quals (n `div` 4)
-- operators
instance QC.Arbitrary Op where
  arbitrary = ops

ops = QC.elements $ [Plus, Minus, Times, Div, Mod, Eq, Less, Greater, In]


newtype Quals = Quals [Qual] deriving (Eq, Show)
instance QC.Arbitrary Quals where
  arbitrary = QC.sized quals

quals 0 = QC.oneof $ [(do (Ident name) <- QC.arbitrary
                      v <- QC.arbitrary
                      return $ Quals $ ((QFor name (Const v):[]))),
                      (do v <- QC.arbitrary
                      return $ Quals $ ((QIf (Const v)):[]))]

quals n = QC.oneof $ [(do (Ident name) <- QC.arbitrary
                      e <- subExp
                      (Quals qs) <- subQuals
                      return $ Quals $ ((QFor name e):qs)),
                      (do e <- subExp
                      (Quals qs) <- subQuals
                      return $ Quals $ ((QIf e):qs)) ]
                      where subQuals = quals (n `div` 2)
                      subExp = expN (n `div` 4)
-- Statements
instance QC.Arbitrary Stmt where
  arbitrary = statements

statements =  QC.oneof $ [(do (Ident name) <- QC.arbitrary
                              exp <- QC.arbitrary
                              return $ SDef name exp),
                      (do exp <- QC.arbitrary
                              return $ SExp exp)]

------ Printing stuff -------
printVal :: Value -> String
printVal NoneVal = "None "
printVal TrueVal = "True "
printVal FalseVal = "False "
printVal (IntVal x) = show x
```

```haskell
printVal (StringVal x) = "'" ++ (escapeStringVal x) ++ "'"
printVal (ListVal x) = "[" ++ (printValueL x True) ++ "]"


escapeStringVal :: String -> String
escapeStringVal "" = ""
escapeStringVal (('\''):rest) = "\\'" ++ (escapeStringVal rest)
escapeStringVal (('\\'):rest) = "\\\\" ++ (escapeStringVal rest)
escapeStringVal (('\n'):rest) = "\\n" ++ (escapeStringVal rest)
escapeStringVal (a:rest) = a : (escapeStringVal rest)

printx :: Eq a => (a -> String) -> String -> [a] -> Bool -> String
printx _ _ [] _ = ""
printx f sep (x:xs) il
  | xs == [] = (f x)
  | otherwise = (if il then (f x) ++ sep else (f x) ++ " ") ++ (printx f sep xs il)

printValueL = printx printVal ", "
printL = printx printExpression ", "
printQualL = printx printQualifier " "
printExpL = printx printStatement ";"

printOperator :: Op -> String
printOperator Plus = "+"
printOperator Minus = "-"
printOperator Times = "*"
printOperator Div = "//"
printOperator Mod = "%"
printOperator Eq = "=="
printOperator Less = "<"
printOperator Greater = ">"
printOperator In = "in "

parens :: Exp -> String
parens e = "(" ++ (printExpression e)  ++ ")"

parens' :: String -> String
parens' e = "(" ++ e  ++ ")"

brackets :: String -> String
brackets e = "[" ++ e ++ "]"

printExpression :: Exp -> String
printExpression (Const val) = printVal val
printExpression (Var name) = name
printExpression (Oper op e1 e2) = (parens e1) ++ (printOperator op) ++ (parens e2)
printExpression (Not e1) = "not " ++ (parens e1)
printExpression (Call name args) = name ++ (parens' (printL args True))
printExpression (List elems) = (brackets (printL elems True))
printExpression (Compr exp quals) = "[" ++ (parens exp) ++ (printQualL quals True) ++ "]"

printQualifier :: Qual -> String
printQualifier (QFor name exp) = "for " ++ name ++ " in " ++(parens exp)
printQualifier (QIf exp) = "if " ++ (parens exp)

printStatement :: Stmt -> String
printStatement (SDef name exp) = name ++ "=" ++ (printExpression exp)
printStatement (SExp exp) = (printExpression exp)

printProgram :: Program -> String
printProgram p = printExpL p True
```