Advanced Programming 2017
Assignment 2

Tom Jager
dgr418@alumni.ku.dk

Tobias Ludwig
fqj315@alumni.ku.dk

September 27, 2017

1 New grammar

First we restructured the given BNF grammar a bit by hand. Note that the structure in our implementation may vary, e.g. with respect to the names of the non-terminals and to the concrete implementation (like the use of `chainl` for left-association, see below).

```
Expr ::= Term1 ',' Expr
       | Term1

Term1 ::= Ident '=' Term1
        | Term2

Term2 ::= Term2 '===' Term3
        | Term2 '<' Term3
        | Term3

Term3 ::= Term3 '+' Term4
        | Term3 '-' Term4
        | Term4

Term4 ::= Term4 '*' Atom
        | Term4 '%' Atom
        | Atom

Atom ::= Number
       | String
       | 'true'
       | 'false'
       | 'undefined'
       | Ident '(' Exprs ')'
       | Ident
       | '[' Exprs ']'
       | '[' ArrayFor ']'
       | '(' Expr ')'

Exprs ::= ϵ
        | Expr1 CommaExprs

CommaExprs ::= ϵ
             | ',' Expr1 CommaExprs

ArrayFor ::= 'for' '(' Ident 'of' Expr1 ')' ArrayCompr

ArrayIf ::= 'if' '(' Expr1 ')' ArrayCompr

ArrayCompr ::= Expr1
             | ArrayFor
             | ArrayIf
```

1. Abolishing **Left-recursion** where unnecessary: this is the case for the definition of `Expr` which consists of two top-level instructions (again `Exprs`!) seperated by comma. We can prevent the left-recursion by just making it right-recursive which is justified, because the comma separator is right-associative (see below).
2. **Precedence** is possible by defining the operators explicitly on different levels which we call `Term1` through `Term4` sticking to the numbering in the task description where level 1 corresponds to the lowest precedence. Because the parser will work through the grammar top-down, we will parse those first. This way we get a hierarchy of operators each of which can only be called with terms from lower levels. Note that, if our hierarchy were that simple we could not use lower precedence level (e.g.

Assignment, Term1) in computations of higher precedence, e.g. in `3 + (x=2)` could not be parsed. However, we include the `'(' Expr ')'` as an option on the `Atom` level thereby closing the circle to the top of the hierarchy. Hence a proper nesting of expressions is possible maintaining a new frame of precedence in every paranthesized expression.

3. **Associativity** for the arithmetic operators comes into the game because we have different levels of terms on either side of an operator. For example, if we want `+` (with precedence 3) to be left-associative, we allow terms of the same precedence only on the left side (making it left-recursive). Thus if there occurs another `+` on the left side this is evaluated first, and on the other hand, if a `+` occurs on the right, this can only be in parantheses. The same goes for `-/*/^` and the opposite, right-recursiveness for the right-associative operators.

2 Parser Combinator Library

2.1 Parsec

In our implementation for the SubScript parser, we decided to utilize the Parsec library. From researching online, we knew that the Parsec library was relatively fast pace and that as we were only parsing simple SubScript programs we would not need a library specifically built for high-speed performance parsing. Therefore, Parsec seemed an acceptable choice as it was relatively user friendly. It also comes equipped with a large range of combinators for parsing strings and spaces which we make use of to construct the parser for SubScript expressions. In particular, Parsec offers the `chainl1` parser, which allows us to preserve left associativity when parsing operations without falling into a loop due to left recursion.

2.2 Try

Parsec also offers the `try` function, which attempts a parser on the input but ensures that if the parser fails then no input is consumed. This allows us to perform backtracking using Parsec. As the `<|>` operator in Parsec is left-biased, the parser will consume inputs when trying the first argument in an expression featuring `<|>`. This is problematic when checking for keywords, looking to see if the program features "," e.t.c. `Try` removes the bias from the alternative operation so that if the first parser fails, no input is consumed ensuring that an unaffected input is available for parsing on the right parser. This allows us to construct a parser with a similar structure to the grammar.

3 Parsers for Number, Ident and String

3.1 Ident

Identifiers have to start with a letter followed by alphanumerics including underscores.

```
identP :: Parser String
identP = do
  c <- letter
  cs <- many (alphaNum <|> satisfy ('_' ==))
  spaces
  if ((c:cs) `elem` keyWords)
    then fail ((c:cs) ++ " is a keyword")
    else return (c:cs)
```

Testing these with QuickCheck yielded 30 passes.

3.2 Number

A number may have at most 8 digits, optionally preceded by a single minus sign. This is done using different parsers for positive and negative numbers.

```
numberP :: Parser Expr
numberP = try negP <|> posP <* many spaces

posP :: Parser Expr
posP =  do
  ns <- many1 digit
  if (length ns < 9)
    then return (Number (read ns))
    else fail "Number too long"

negP :: Parser Expr
negP = do
  x <- string "-"
  ns <- many1 digit
  if (length ns < 9)
    then return (Number (read (x ++ ns)))
    else fail "Number too long"
```

This also passed 100 QuickCheck test, producing a number $-10^9 < x < 10^9$.,

3.3 String

SubsScript Strings are enclosed in single quotes and can include escapes. We wrote a function `stringP` that parses the beginning of a string, the opening single quote, and then calling `substringP` as often as possible. `substringP` consumes a character and returns it, unless it was a backslash, in which case it consumes another character. Depending on what the escaped character is it returns an error "unknown escape sequence" (which is not displayed) or the valid corresponding Haskell escape sequence.

For the end of the string we use a special parser because we don't want to stop at an escaped single quote which would be the case if used just `manyTill substringP string "'"`. Instead we have a function `endstringP` which ensures that no escaping backslash precedes the closing single quote. Sadly that part does not work.

```haskell
stringP :: Parser Expr
stringP = do
  string "'" -- start of string
  c <- manyTill substringP endstringP
  return (String (concat c))

endstringP :: Parser ()
endstringP = do
  notFollowedBy (char '\\')
  string "'"
  return ()

substringP :: Parser String
substringP = do
  c1 <- anyChar
  case [c1] of
    "\\" -> do               -- single backslash escapes the next char
        c2 <- anyChar        -- which requires reading
        case c2 of
          '\n' -> return ""
          '\\' -> return "\\"
          '\'' -> return "'"
          'n'  -> return "\n"
          't'  -> return "\t"
          _    -> fail "unknown escape sequence"

    _ -> return [c1]      -- otherwise, just return the char
```

3.4 String Testing

We tested `stringP` by writing a generator called `NoEscape` for Haskell-Strings including single quotes but without escapes. We prepend a single quote to the generated strings (because we want them to start with and filter those which actually start by a single quote and include another one. However we realized that there is a bug in the test code itself, so `quickCheck stringP` will always fail after a few trials.

But we do have some confidence that Strings are parsed rightly (at least in the unescaped version) as many handwritten tests and the online TA confirms. However the escape sequences are known issues. Escaping accepts any Haskell-escapes so for example not only `\n` but also `\a` which is illegal in subscript. The escaped single quote and the escaped newline do not work either. We had no more time to debug at this point.

4 Whitespace handling

Whitespace is always parsed after a token. It may follow each operator, atom and other symbol (like keywords, square brackets and parentheses).

We wrote a function `symbolP` that is called after every terminal, parsing a certain String (using Parsec's `string`) and the following whitespace. Technically, it can be easily collected with the Parsec function `spaces` which skips zero or more whitespace characters. We also have to parse whitespace after Numbers in order to prevent something like `3s` or even `3.14` (we don't care about floats!).

Also comments are regarded as whitespace, which are initiated by double slash. So we wrote a unifying `discard :: Parser ()` function that just neglects its input, calling either the Parsec function `space` or our `commentP`.

```haskell
commentP :: Parser ()
commentP = do
  symbolP "//"
  manyTill anyChar (string "\n")
  return ()
```

4.1 Testing Whitespace

We wrote two non-automized tests to show whitespace is parsed and ignored. The one tests, if `1234z` fails, and succeeds. The other one tests, that whitespace in strings is not ignored, also succeeds.

Another test `testComments` also succeeds if the input contains a comment ended by `\n`. However there is one known issue from the online TA that the comments do not work as token separators, e.g. in For.

5 Operators

5.1 Comma

As each SubScript program is treated as an expression, the parser for SubScript is the expression parser `exprP`. Programs can either be multiple expressions or a single expression. Programs with multiple expressions are formed using the `","` operator which has the weakest precedence and so will be applied last. This is implemented by trying to parse all other operators between expressions first and only if they all fail do we parse the `","`. If there is only a single expression and no `","` `exprP` performs the single expression parser `expr1P`.

Comma expressions are parsed using `commaP` a which parses and binds a single expression using `expr1P`. It then consumes the token `","` and then performs `exprP` binding its value. A parser returning a `Comma` expression constructed with the single expression and the other expression. As `exprP` is higher in the grammar of the language, it ensures that comma will parse the right hand side of the comma first and so is right associative.

```
exprP :: Parser Expr
exprP = try commaP
      <|> expr1P
```

```
commaP :: Parser Expr
commaP = do
  expr1 <- expr1P
  symbolP ","
  expr2 <- exprP
  return (Comma expr1 expr2)
```

5.2 Assign

The operation of next precedence is the `"="` operator for assignment. As such, the assign parser is placed next in the hierarchy. `expr1P` tries the assignment parser and if it fails it drops down to the next step in the hierarchy, the Term 2 parser. `assignP` parses and binds an identifier, then consumes the `"="` token followed by binding a single expression parsed by `"expr1P"`. A parser for `Assign` expressions constructed by the identifier and the expression is returned. To ensure that `"="` is right associative the expression on the right hand side is parsed by a parser higher in the hierarchy.

```
assignP :: Parser Expr
assignP = do
  name <- identP
  symbolP "="
  expr <- expr1P
  return (Assign name expr)
```

```
expr1P :: Parser Expr
expr1P = try assignP
       <|> term2P
```

5.3 Left Associative Operators

The remaining operators are all parsed in a similar manner. In order to maintain their precedence, the `"==="` and `"<"` parsers are implemented first, followed by `"+"` and `"-"`, followed by `"*"` and `"%"`. Each operator is parsed in a similar way, the `chainl1` parser combinator is used to parse at least 1 occurrences of the tier below. `chainl1` ensures that each operation is parsed as left associative and avoids being stuck in a recursive loop with nothing being consumed. This combinator works by taking a parser for each term of the operation and another parser for a function that takes both terms and returns the resultant expression. This function parser `compOp`, `addOp` and `multOp` parse their corresponding operators and use `binOp` to return a function that constructs the appropriate `Call` expression from two other expressions. By ensuring that each term of `chainl1` is lower down in the hierarchy, it is ensured that operations lower down in the hierarchy are performed first.

```
term2P :: Parser Expr
term2P = term3P `chainl1` compOpP

term3P :: Parser Expr
term3P = term4P `chainl1` addOpP

term4P :: Parser Expr
term4P = atomP `chainl1` multOpP


compOpP :: Parser (Expr -> Expr -> Expr)
compOpP = do{ s <- symbolP "==="; return (binOp s)   }
       <|> do{ s <- symbolP "<"; return (binOp s) }
```

```
addOpP :: Parser (Expr -> Expr -> Expr)
addOpP = do{s <- symbolP "+"; return (binOp s)    }
       <|> do{s <- symbolP "-"; return (binOp s) }

multOpP :: Parser (Expr -> Expr -> Expr)
multOpP = do{s <- symbolP "*"; return (binOp s)    }
       <|> do{s <- symbolP "%"; return (binOp s) }

binOp :: String -> Expr -> Expr -> Expr
binOp op expr1 expr2 = Call op [expr1,expr2]
```

5.4 Operator Testing

Testing begins with `operatorTest`, where an example expression of each operator was tested. All operators passed. The testing values can be seen here and the tests repeated by calling `operatorTest` from Parser/Tests.hs

```
opSubsList = ["5,x,4","x=6","4 < 5","4 === 5", "8+3","7-6","8*3","8%3"]

opExprList = [Comma (Number 5) (Comma (Var "x") (Number 4)),Assign "x" (Number 6), Call
↪  "<" [Number 4,Number 5], Call "===" [Number 4,Number 5], Call "+" [Number 8, Number
↪  3], Call "-" [Number 7, Number 6], Call "*" [Number 8, Number 3], Call "%" [Number 8,
↪  Number 3]]
```

Next the precedence properties are tested. This is done by writing a single expression containing all the operators and ensuring that they bind in the correct order. The test passed and can be recreated by calling `opPrecTest`

```
precSubsList = ["1,x=2+3-4*5%6"]

precExprList = [Comma (Number 1) (Assign "x" (Call "-" [(Call "+" [Number 2,Number
↪  3]),(Call "%" [(Call "*" [Number 4,Number 5]),Number 6])]))]
```

Finally the associativity of each operator is tested. The test string in this case involves using each operation twice in an expression and ensuring they bind in the correct order. All tests passed and can be recreated using `opAssocTest`

```
assocSubsList = ["a,b,c","a=b=c","a+b+c","a-b-c","a*b*c","a%b%c"]

assocExprList = [Comma (Var "a") (Comma (Var "b") (Var "c")), Assign "a" (Assign "b" (Var
↪  "c")), Call "+" [Call "+" [Var "a",Var "b"],Var "c"], Call "-" [Call "-" [Var "a",Var
↪  "b"],Var "c"], Call "*" [Call "*" [Var "a",Var "b"],Var "c"], Call "%" [Call "%" [Var
↪  "a",Var "b"],Var "c"]]
```

6 Atoms

6.1 Simple Atoms

In our grammar, the expressions that do not rely on operations have been termed Atoms as they require no further division. These are all types of expressions other than `Comma`, `Assign` and arithmetic operations. The Atoms; `Number` and `String` are parsed by using the corresponding number and string parsers discussed earlier. `TrueConst`, `FalseConst` and `Undefined` are parsed by using `symbolP` to parse the tokens 'true', 'false' or 'undefined' and returning the corresponding expression.

`Var` expressions are parsed by performing the `identP` to obtain the variable name and then returning the resultant `Var` expression.

Function calls are parsed in a similar manner but require a `"("` token followed by a list of single expressions separated by commas. This list is parsed and bound by `exprsP` (discussed in 5.3 Arrays). `callP` then consumes a `")"` token and returns a call constructed from the identifier and the list of expressions. As `"varP"` would parse correctly and halt after parsing an identifier, it is important to attempt `callP` first to check whether the identifier is for a function name or a variable name.

The last simple atom are the functions between parentheses which is dealt with by `parensP`. This parser consumes a `"("` token followed by binding an expression parsed by the `exprP` parser. It then consumes a closing `")"` token and returns the expression as a single expression.

```
trueP :: Parser Expr
trueP = do
  symbolP "true"
  return TrueConst

falseP :: Parser Expr
falseP = do
  symbolP "false"
  return FalseConst
```

```
undefP :: Parser Expr
undefP = do
  symbolP "undefined"
  return Undefined

varP :: Parser Expr
varP = do
  name <- identP
  return (Var name)
```

```
callP :: Parser Expr
callP = do
  name <- identP
  symbolP "("
  exprs <- exprsP
  symbolP ")"
  return (Call name exprs)

parensP :: Parser Expr
parensP = do
  symbolP "("
  expr <- exprP
  symbolP ")"
  return expr
```

6.2 Array Comprehensions

Array atoms can be split into two parts, ones constructed by array comprehensions and ones that are simple arrays. The ones constructed by array comprehensions parse a `"["` token and then extracts an `ArrayCompr` by performing the `arrayForP` parser. This uses `symbolP` to consume tokens for `"for"` and `"("`. Then an identifier is parsed and extracted. The token `"of"` is then parsed before a single expression is parsed and extracted. Finally, a `")"` token is parsed and `arrayCompP` is used to parse and bind an `ArrayCompr` value. `arrayForP` concludes by returning an `ACFor` value constructed from the identifier, single expression and the array comprehension.

The parser for array comprehensions, `arrayCompP`, simply attempts to parse a for comprehension, if comprehension or a body comprehension until one of them succeeds. Thus use of the try function prevents inputs from being consumed on a failing parser.

Lastly, `arrayIfP` operates by consuming tokens for `"if"` and `"("` followed by parsing and binding a single expression. It then consumes a `")"` token followed by parsing and binding another array comprehension. `arrayIfP` then returns an `ACIf` data type constructed from the single expression and the array comprehension.

```
arrayIfP :: Parser ArrayCompr
arrayIfP = do
  symbolP "if"
  symbolP "("
  expr <- expr1P
  symbolP ")"
  compr <- arrayCompP
  return (ACIf expr compr)

arrayCompP :: Parser ArrayCompr
arrayCompP = try arrayForP
         <|> try arrayIfP
         <|> do
           expr <- expr1P
           return (ACBody expr)
```

```
arrayForP :: Parser ArrayCompr
arrayForP = do
  symbolP "for"
  symbolP "("
  name <- identP
  string "of"
  discard1
  expr <- expr1P
  symbolP ")"
  array <- arrayCompP
  return (ACFor name expr array)
```

6.3 Arrays

Arrays are parsed by `arrayP`. This function parses and binds a list of single expressions in between the `"["` and `"]"` tokens. The list of expressions is parsed by `exprsP` which uses the `sepBy` combinator to apply the `expr1P` 0 or more times to input separated by `","`. This returns a parser for a list of expressions.

```
arrayP :: Parser Expr
arrayP = do
  symbolP "["
  exprs <- exprsP
  symbolP "]"
  return (Array exprs)
```

```
exprsP :: Parser [Expr]
exprsP = expr1P `sepBy` (symbolP ",")
```

6.4 `atomP`

Each of the above parsers are combined into a single parser for Atoms, `atomP`. This uses the try function and the `<|>` to attempt to apply each parser until one succeeds. The try function ensures that if a parser fails it does not consume any input so the stream is unchanged for the next parser to be attempted.

```
atomP :: Parser Expr
atomP = try numberP
      <|> try stringP
      <|> try callP
      <|> try varP
      <|> try trueP
      <|> try falseP
      <|> try undefP
      <|> try (do
        symbolP "["
        compr <- arrayForP
        symbolP "]"
        return (Compr compr))
      <|> try arrayP
      <|> parensP
```

6.5 Atom Testing

To test the atoms I have written a list of SubScript expressions as strings and a list of their corresponding expressions. By running atomTest from the Parser/Tests.hs file you parse each SubScript string and compare the parsed version against the expression. All atoms passed their tests.

```
List of SubScript expressions as strings
["5764","'hello world'","Array(0,1,1)","xs","true","false","undefined",
"[ for (x of xs) if (x === 5) x ]","['h','e','l']","(x=y,z=x)"]

List of the corresponding Expressions
[Number 5764,String "hello world",Call "Array" [(Number 0),(Number 1),(Number 1)],
Var "xs", TrueConst, FalseConst,
Undefined, Compr (ACFor "x" (Var "xs") (ACIf (Call "===" [(Var "x"),(Number 5)])
(ACBody (Var "x")))), Array [(String "h"),(String "e"), (String "l")],
Comma (Assign "x" (Var "y")) (Assign "z" (Var "x"))]
```

7 Conclusion

We wrote a SubScript parser using Parsec which was rather straight forward because of the parser combinators already offered by the rich library. The parser is pretty much working, the only known issues (also reflected in the Online TA) are occurring in strings, especially the escaping whenever it differs from Haskell's defaut. Also, as a minor issue, separation of tokens by comments does not work.