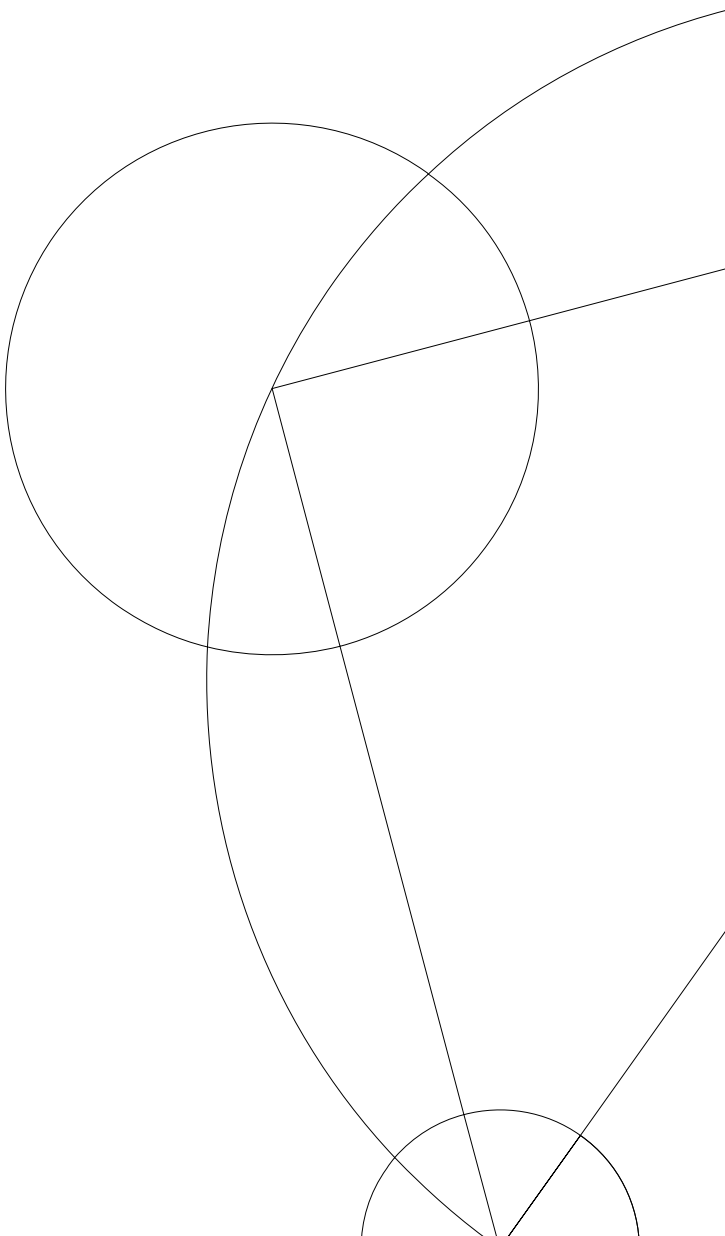
UNIVERSITY OF COPENHAGEN

# AP Assignment 2 - SubScript Parser

DIKU University of Copenhagen 2017

Jan Sokol, Denis Trebula
{kgj360,jmp640}@alumni.ku.dk

September 27, 2017



# 1 Subscript Parser

# 2 Choice of parser combinatory library

At first we had to choose a solid library for working with parsing and grammar in our project. Two main libraries which got our attention were Parsec and ReadP. After further decision making we chose to use the Parsec combinatory library because according to the documentation it performs best on predictive (LL) grammars. In addition, it contains extensive error messages which make the development of the code easier. Finally, we can combine small parsing functions to build more complex parsers.

# 3 Usage of biased combinators

When we need to parse a text for example basic true or false text, we try applying it to a parser function. If we didnt use the try combinator, we could lose the already parsed text and none of the other parsers would have given a correct result. With the try combinator, we have the ability to backtrack and pass the same text to another parser. This is needed when we parse the SubScript language because many of its grammar productions have many alternatives. If a parser fails for an one alternative, then we have the ability to try the same text for the next alternative.

# 4 Grammar

To have grammar that is working with Parsec library, we have to remove left recursion. If a context-free grammar has a $A \rightarrow A\alpha$ rule, then is left-recursive. In grammar given in assesement, we can see this in first rule for example:

```
Expr ::= Expr , Expr | Expr1
```

To remove left recursion from grammar, we need to replace left-recursive rules.[1][2] Rule $A \rightarrow A\alpha|\beta$ can be rewritten as:
$A \rightarrow \beta A'$
$A' \rightarrow aA'$

Also as stated in the assesement, some operators precedence the others. For example `*, %` have higher precedence than `+, -`, so we split `Expr1` rule into multiple rules.

Using this we transformed grammar from assessement to:

```
Expr        ::= Expr1 Expr'

Expr'       ::= ',' Expr | eps

Expr1       ::= Ident '=' Expr1
            | Expr2

Expr2       ::= Expr3 Expr2'

Expr2'      ::= '===' Expr3 Expr2'
            | '<' Expr3 Expr2'
            | eps

Expr3       ::= Expr4 Expr3'
```

```
Expr3'    ::= '+' Expr4 Expr3'
            | '-' Expr4 Expr3'
            | eps

Expr4       ::= ExprTerm Expr4'

Expr4'    ::= '*' ExprTerm Expr4'
            | '%' ExprTerm Expr4'
            | eps

ExprTerm  ::= Number
            | String
           | 'true'
           | 'false'
           | 'undefined'
           | Ident ExprIdent
           | '[' Exprs ']'
           | '[' 'for' '(' Ident 'of' Expr1 ')' ArrayCompr Expr1 ']'

ExprIdent ::= '(' Exprs ')'
            | eps

Exprs       ::= Expr1 CommaExprs
            | eps

CommaExprs  ::= ',' Expr1 CommaExprs
            | eps

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

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

ArrayCompr  ::= Expr1
            | ArrayFor
            | ArrayIf

Ident       ::= (as in assignment)
Number      ::=
String      ::=
```

# 5 Whitespace handling

For optimization purposes we handle whitespaces using three functions. To remove any unnecesary whitespaces around string token we use function `sToken` that takes string as a parameter. (Function uses `spaces` parser from Parsec library and removes any amount of whitespaces, that would make parser fail otherwise.)

```
sToken :: String -> Parser String
sToken s = try (spaces >> string s <* spaces)
```

Second function `whitespaceParser` is more robust, also calls funtion `commentHandler` if necesary. Is used mainly in function `backslashCheck` that parses strings. For easier measures

we also used function whitespaceHandler which can be found in slides from second lecture on parser.

```
whitespaceParser :: Parser ()
whitespaceParser = do
                    first <- many $ oneOf " \n\t"
                    second <- many commentHandler
                    when (not (null first) || not (null second)) $
                      do whitespaceParser -- recursive call
                         return ()
```

# 6 Parser implementation and Assesment

For implementation we used skeleton from the assesment zip file. We changed type of ParserError from one defined in assesment to one in `Text.Parsec.Error` library.

Based on our tests and OnlineTA we can say that our parser is able to succesfully parse SubScript files given in the assesment directory. Parser only has problems with comments, if they are inside string (such as name of a variable).

Precedence of operators is correctly implemented (acording to our tests and tests from OnlineTA), also operators have left association (we have included tests for this). If a number has more than 8 digits (or 7 with negative sign), parser won't accept it.

# 7 How to run code and reproduce test results

We have managed to implement a bit of unit testing in the `Parser/Tests.hs`. We used tasty framework which was widely recommended by the Haskell community. Take in mind that in order to properly compile and run tests, you must have downloaded and installed tasty framework and its plugin, `tasty-HUnit`. In order to do so type this in command line:

```
stack install tasty
stack install tasty-hunit
```

To run the code with our unit tests:

```
stack exec ghci -- -W Parser/Tests.hs
```

And run the main function as:

```
main
```

This function will perform all the tests in the Tests.hs module. The tests evaluate a given input to a parser if it is the same as the expected output. If the result matches the expected value, then the tests are successful.

## 7.1 Resources

[1] http://www.csd.uwo.ca/ moreno//CS447/Lectures/Syntax.html/node8.html
[2] http://web.eecs.umich.edu/ weimerw/2009-4610/reading/left-recursion.pdf