Assignment 2: A SubScript Parser

Advanced Programming, fall 2016

Nikolaj Høyer
ctl533@ku.dk

Andrew Tristan Parli
qcm239@ku.dk

1. CHOICE OF PARSER COMBINATOR LIBRARY

We focused on looking at the two main libraries presented in the project text, namely Parsec and ReadP. ReadP is a parser generator library that is already part of Haskell's standard library. Parsec is a more industrial strength "batteries included" parser combinator library that is not part of Haskell's standard library.

Since we are not allow to use Parsec's power tools for this assignment, we decided to use ReadP, as it has less documentation to digest. Furthermore, since ReadP is part of Haskell's base, it is perhaps less likely to cause problems in the future. Additionally, the conceptually usage of ReadP can be applied easily to the usage of "plain" Parsec.

2. GRAMMAR REVISIONS

In order to deal with the precendence and associativity of the operators in SubScript from Table 1 in the assignment text, we revised the grammar by introducing some new terminals and non-terminals. For example in order to deal with the left associativity of the operators for expressions, we replaced

$$\begin{array}{lcl} Expr & ::= & Expr\ \texttt{','}\ Expr \\ & | & Expr1 \end{array}$$

with

$$\begin{array}{lcl} Expr & ::= & Expr\texttt{','}Expr1 \\ & | & Expr1 \end{array}$$

Then in order to deal with the precedence of the operators, we needed to add more non-terminals, while respecting the left associative property of the operators. For example, in order to deal with `'+'` and `'-'` having lower precedence than `'*'` and `'%'`, we have done the following,

$$\begin{array}{lcl} Expr4 & ::= & Expr4\texttt{'+'}Expr5 \\ & | & Expr4\texttt{'-'}Expr5 \\ & | & Expr5 \end{array}$$

$$\begin{array}{lcl} Expr5 & ::= & Expr5\texttt{'+'}Expr6 \\ & | & Expr5\texttt{'-'}Expr6 \\ & | & Expr6 \end{array}$$

The revised grammar is shown in Table 1.

3. HANDLING WHITESPACE

Whitespace is handled by the following helper function:

```
1 token :: ReadP a -> ReadP a
2 token p = skipSpaces >> p
```

, which utilizes the `skipSpaces` function of the ReadP module to ignore whitespace while parsing.

4. EXTENT OF SUBSCRIPT IMPLEMENTATION

First off, `parseString` is not yet implemented, so the entry point to any parsing is the `pStms` function. Second, we had issues with ambiguity in the recursive parts of *Expr*, as testing produced a lot of itentical results even for an expression like `3 + 3`. Third, we have left array comprehensions out of the implementation for now, and array declarations does not seem to function as intended.

We have, however, implemented parsers for most of the remaining grammar: statements, variable declaration, terminal expressions, parentheses and function calls.

5. TESTING

We have managed to implement a bit of unit testing in the `Parser/Tests.hs` file, however automating the process has been touchy between our systems. From the `src` folder in the terminal, one should be able to use the follow to run the unit tests,

```
1 stack exec runhaskell -- Parser/Tests.hs
```

Additionally, we used the following function to test parsers in `ghci` on the fly.

```
1 readP_to_S (pStms <* (skipSpaces >> eof)) "1+1"
```

Table 1: Grammar with associativity and precedence

Program	::=	*Stms*
Stms	::=	ϵ
	|	*Stm* `';'` *Stms*
Stm	::=	`'var'` *Ident AssignOpt*
	|	*Expr*
AssignOpt	::=	ϵ
	|	`'='` *Expr*1
Expr	::=	*Expr* `','` *Expr*1
	|	*Expr*1
*Expr*1	::=	*Ident AfterIdent*
	|	*Expr*2
*Expr*2	:=	*Expr*2 `'==='` *Expr*3
	|	*Expr*3
*Expr*3	:=	*Expr*3 `'<'` *Expr*4
	|	*Expr*4
*Expr*4	:=	*Expr*4 `'+'` *Expr*5
	|	*Expr*4 `'-'` *Expr*5
	|	*Expr*5
*Expr*5	:=	*Expr*5 `'*'` *Expr*6
	|	*Expr*5 `'%'` *Expr*6
	|	*Expr*6
*Expr*6	:=	*Number*
	|	*String*
	|	`'true'`
	|	`'false'`
	|	`'undefined'`
	|	`'['` *Exprs* `']'`
	|	`'['` `'for'` `'('` *Ident* `'of'` *Expr*1 `')'` *ArrayCompr Expr*1 `']'`
	|	`'('` *Expr* `')'`
AfterIdent	::=	ϵ
	|	`'='` *Expr*1
	|	*FunCall*
FunCall	::=	`'.'` *Ident FunCall*
	|	`'('` *Exprs* `')'`
Exprs	::=	ϵ
	|	*Expr*1 *CommaExprs*
CommaExprs	::=	ϵ
	|	`','` *Expr*1 *CommaExprs*
ArrayCompr	::=	ϵ
	|	`'if'` `'('` *Expr*1 `')'` *ArrayCompr*
	|	`'for'` `'('` *Ident* `'of'` *Expr*1 `')'` *ArrayCompr*