Assignment 2: A SubScript Parser

Based on a task from the 2015/16 final exam

Version of 19th September 2017

Last week, you got acquainted with the semantics of SubScript. The relationship between SubScript and JavaScript array-comprehensions was perhaps a bit tenuous, as we presented examples in JavaScript-like syntax, but you could only type in a corresponding SubScript abstract syntax tree.

Your task this week is to write a *parser* for SubScript.

We refer you to last week's assignment for sample SubScript programs.

The handout for this assignment contains a skeleton for both this assignment and the previous. You can incorporate your solutions to both into a full-blown SubScript parser and interpreter at your leisure (e.g., as exam reading). This is *not* part of the assignment.

The handout organization gives you a hint on how you can test module internals in a separate file, without exposing them, while still meeting the API requirements below.

SubScript Abstract Syntax Tree

The SubScript abstract syntax tree is defined in the handed out `SubsAst.hs`. We list this module below for quick reference. You should not change these types, but merely import and work with the `SubsAst` module.

```
module SubsAst where

data Expr = Number Int
          | String String
          | Array [Expr]
          | Undefined
          | TrueConst
          | FalseConst
          | Var Ident
          | Compr ArrayCompr
          | Call FunName [Expr]
          | Assign Ident Expr
          | Comma Expr Expr
         deriving (Eq, Read, Show)

data ArrayCompr = ACBody Expr
                | ACFor Ident Expr ArrayCompr
                | ACIf Expr ArrayCompr
                deriving (Eq, Read, Show)

type Ident = String
type FunName = String
```

SubScript Parser

The grammar of SubScript is specified below. Keywords and special symbols are written between single quotes, and ϵ represents an empty string.

To ease development, we omit many nuances of JavaScript from SubScript. In the grammar below you will notice that many constructions valid in JavaScript are not valid in SubScript.

Expr	::=	*Expr* ‘`,`’ *Expr*
	|	*Expr1*
Expr1	::=	*Number*
	|	*String*
	|	‘`true`’
	|	‘`false`’
	|	‘`undefined`’
	|	*Ident*
	|	*Expr1* ‘`+`’ *Expr1*
	|	*Expr1* ‘`-`’ *Expr1*
	|	*Expr1* ‘`*`’ *Expr1*
	|	*Expr1* ‘`%`’ *Expr1*
	|	*Expr1* ‘`<`’ *Expr1*
	|	*Expr1* ‘`===`’ *Expr1*
	|	*Ident* ‘`=`’ *Expr1*
	|	*Ident* ‘`(`’ *Exprs* ‘`)`’
	|	‘`[`’ *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*

Ident	::=	(see below)
Number	::=	(see below)
String	::=	(see below)

Note that this grammar specifies that a syntactically well-formed array comprehension must start with a `for`-clause. The more liberal syntax in the `Expr` AST would correspond to having the penultimate production for *Expr1* read, instead:

| ‘`[`’ *ArrayCompr* ‘`]`’

However, such a grammar would leave it ambiguous whether, e.g., the text “`[ 5 ]`” should be parsed as `Array [Number 5]`, or as `Compr (ACBody (Number 5))`. Though these actually have the same *meaning* (i.e., both should evaluate to `ArrayVal [Intval 5]`), a proper SubScript syntax specification should still designate only one of them as the correct parse. (The grammar above also contains other ambiguities, which will be resolved below.)

Lexical specification

The three grammar symbols not defined in the grammar are described as follows:

- *Ident* : JavaScript identifiers have a very liberal syntax. For simplicity, we stay rather more conservative: a SubScript *Ident* consists of an upper- or lowercase English letter ('A' through 'Z'), followed by zero or more letters, digits, or underscores.

  However, an identifier cannot be one of the keywords otherwise used in SubScript, i.e. `true`, `false`, `undefined`, `for`, `of`, or `if`. Using other JavaScript keywords is allowed in SubScript, but this might hamper the testability of your scripts.

- *Number* : All numbers in JavaScript are IEEE-754 double-precision floating point numbers (doubles). Dealing in doubles is beyond the scope of SubScript: A *Number* consists of 1–8 decimal digits, optionally preceded by a single minus sign (without any space between the sign and the first digit).[1] Note that the sign is considered a part of the *Number* token, not a separate prefix operator, so that, e.g., "`-x`" would not be a well-formed expression. Also, `+` *cannot* be used as a sign.

- *String* : A SubScript string constant consists of zero or more *string characters* enclosed between single quotes (`'···'`). Allowed string characters are all printable ASCII characters (so excluding newlines and tabs), except single quotes and backslashes. Additionally, the following two-character sequences are allowed inside strings:

  - `\'` : represents a single-quote character.
  - `\n` : represents a newline character
  - `\t` : represents a tab
  - `\\` : represents a (single) backslash character
  - `\`*newline* (i.e., a backslash followed by a single newline character): ignored, to allow string constants to be written over multiple lines. For example, the SubScript program

    ```
    x = 'foo\
    bar'
    ```

    will set the variable `x` to the 6-character string "`foobar`".

  All other combinations starting with a backslash are illegal.

Whitespace rules

SubScript program texts may only contain printable characters (letters, numbers, punctuation symbols, and spaces), newline characters (`\n`) and tabulation characters (`\t`). Syntactic tokens may be separated by zero or more whitespace characters (spaces, newlines, or tabs).

[1] Every integer in the range $[-\underbrace{99999999}_{8\text{ digits}};\underbrace{99999999}_{8\text{ digits}}]$ is both exactly representable as an IEEE-754 double-precision floating point number, and fits in a Haskell `Int`.

However, at least one whitespace character is needed to separate keywords from adjacent alphanumeric characters and underscores; e.g., `true3` would be parsed as an identifier, rather than the keyword `3` followed by the number `3`.

Additionally, SubScript programs may contain *comments*, which start by "`//`" (two consecutive slashes) and run until the end of the line. Comments are also considered white space, and may hence be used to separate token where this is required. Comments are not allowed inside string constants (i.e., any slashes inside strings will just be considered part of the string itself, without any special significance).

Operator disambiguation

Table 1 presents the precedences and associativity of the operators in SubScript. Note that these do not necessarily agree with the corresponding operators in full JavaScript.

Precedence	Operator(s)	Associativity
4	'*' '%'	left
3	'+' '-'	left
2	'===' '<'	left
1	'='	right
0	','	right

Table 1: Operator precedence and associativity in SubScript.

What to implement

You should implement a module `SubsParser` with the following interface.

A function `parseString` for parsing a SubScript expression given as a string:

```
parseString :: String -> Either ParseError Expr
```

where you decide and specify what the type `ParseError` should be; the only requirement is that it must be an instance of `Show` and `Eq`. The type `ParseError` must also be exported from the module. The handed-out skeleton code already has the exports set up correctly.

Likewise, you should implement a function `parseFile` for parsing a SubScript program given in a file located at a given path:

```
parseFile :: FilePath -> IO (Either ParseError Expr)
```

Where `ParseError` is the same type as for `parseString`.

As you can see, there are no elements of the abstract syntax tree for representing the arithmetical operators '+', '-', '*', or '%', nor is there any dedicated way of representing the relational operators '<' or '==='. These are instead represented as calls to built-in

functions. For instance, the SubScript expression `38 + 4` would be represented as the `Expr` value `Call "+" [Number 38, Number 4]`.

You may use either ReadP, Parsec, or any parser-combinator library handed out at the lectures. You will find Haskell skeletons for the parser and abstract syntax trees on Absalon.

If you use Parsec, then only plain Parsec is allowed, namely the following submodules of `Text.Parsec`: `Prim`, `Char`, `Error`, `String`, and `Combinator` (or the compatibility modules in `Text.ParserCombinators`), in particular you are *disallowed* to use `Text.Parsec.Token`, `Text.Parsec.Language`, and `Text.Parsec.Expr`.

Together with your parser you must also hand in a test-suite to show that your parser works (or where it does not work). That is, that it correctly parses valid programs and rejects invalid programs.

Advice for your solution

1. Start by transforming the grammar to meet the precedence and associativity rules.
2. Write down the final grammar in your report.
3. Choose a parser combinator library and argue for your choice in your report.
4. Write parsers for *Ident* , *Number* , and *String* . Test these.
5. Write a parser for the operator-free parts of the grammar, with proper whitespace (including comments) skipping. Test this.
6. Extend the parser with the operators, respecting their associativity and precedence.

If you do not make it the whole way through the above plan, clearly state which *subset* of the grammar you think your parser supports. Explain why the disallowed language constructs cause you problems. If you wrote tests along the way, this subset is already tested, and you are ready to assess your code.

Consider making more elaborate tests, e.g. using property-based testing.

In either case, state the properties that you think your tests test.

A Subs program

We also hand out a `Subs.hs` with a `main` function. Once you have something that begins to look like a parser, you can test it as follows:

```
$ stack exec runhaskell -- -W Subs.hs -p intro.js
```

This should also indicate how you can write your own (smaller) test programs.

What to hand in

You should hand in two things:

1. A short report, `report.pdf`, explaining your code, and containing an assessment of your implementation, including what this assessment is based upon.

   In the report, you should also:

   a Justify your choice of parser combinator library.

   b If you use any *biased* combinators, such as `try` in Parsec, or `<++` in ReadP, discuss why you can't make due with unbiased combinators.

   c Present the final, transformed grammar, corresponding to your implementation.

   d Discuss how you handle whitespace.

   e Discuss in-how-far your parser can parse SubScript scripts.

   f Tell us how we can run your code and reproduce your test results.

2. A ZIP archive `src.zip`, containing one directory `src`, containing your source code and tests. It should be possible to run your source as submitted.

To keep your TA happy, follow these simple rules:

1. Clean up your code before you submit.
2. Test your code before you submit.
3. Comment the non-trivial parts of your code.
4. Your module should not export partial functions.
5. GHC(i), invoked with `-W`, should not yield any errors or warnings for your code.
6. `hlint` should not yield any hints for your code.