Assignment 2: A SubScript Parser

Based on a task from the 2015/16 final exam

Version 1.0

Due: Wednesday, September 26 at 20:00

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 ASCII 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: here, 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 `true` 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 tokens 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 associativities 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.

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]`.

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, the module 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`. (We have already provided a suitable implementation of this function.)

You may use either ReadP or Parsec for your implementation. 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

We recommend that you proceed roughly as follows:

1. Choose a parser combinator library (ReadP or Parsec) and briefly motivate your choice in the report.
2. Write rudimentary parsers for *Ident* (for now, not prohibiting keywords), *Number* (ignoring the length restrictions), and *String* (not handling backslash-escapes). Test these. It's OK if some of the more esoteric test cases still fail at this stage.
3. Write a parser for the operator-free parts of the grammar, including *systematic* whitespace- (but possibly not yet comment-) skipping. Test this.
4. Add minimal operator parsing, not yet worrying about getting precedence and associativity completely right. This should allow you to parse all "nice" SubScript programs, possibly after adding a few extra parentheses.
5. Fix the operator parsing to properly reflect precedences and associativities. Test the result thoroughly.
6. Add/fix all the little things: restrictions on number constants, escapes in string constants, comment-skipping, variables immediately next to keywords, etc. Test each. If some feature seems excessively hard to get right, move on to the next one.
7. Clean up your parsing code. If you use Parsec, justify (with a comment) each time you need to use `try`. (Your code should contain few, if any, uses of `try`, since SubScript – like full JavaScript – is meant to be parseable without lookahead/backtracking.) If you use ReadP, likewise justify any use of *biased* combinators (like `<++` or `munch`): are they needed for correctness or (substantial) efficiency improvement? Make sure your cleanups doesn't break any tests!

If you do not make it the whole way through the above plan, clearly state which aspects of the grammar you think your parser supports in full, partly, or not at all. Leave in any failing test cases, to demonstrate that you at least understand what the correct outcome would be.

If time permits, consider making more elaborate tests, e.g., using property-based testing. Be sure to explain what properties your tests are testing, and why.

A parser/interpreter program

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

```
$ stack exec runhaskell -- -W Main.hs -p ../examples/intro.js
```

(If you omit the `-p`, the tool will also attempt to run the interpreter after parsing.)

What to hand in

Code

Form For improving testability while maintaining clean APIs, we have adjusted the organization of the source code to facilitate unit testing of module-internal functionality. The module `SubsParser` (file `SubsParser.hs`) now only presents the external interface of the parser, with just the main parsing functions exposed; you should *not* modify this file. Rather, your parser implementation code should be added in the module `Parser.Impl` (file `Parser/Impl.hs`), from which `SubsParser` imports.

Modules *using* the parser (including the sample command-line tool) should still import `SubsParser`, as should *black-box testing* (i.e., tests that would make sense no matter how the parser is implemented). On the other hand *white-box tests*, which may test individual sub-parsers and other auxiliary functions (and are thus specific to your particular code), can import the full `Parser.Impl` directly. The handout directory has everything set up correctly.

Reminder: We are moving to a Stack-project-compatible organization of handouts/handins. In the provided `handout.zip`, you will find a single directory `handout`, which contains the Haskell source files in a subdirectory `src`, the sample SubScript programs from the assignment text in subdirectory `examples`, and some very rudimentary tests in subdirectory `tests`, as well as some Stack metadata. See the `README` file for details. Your submission should follow the *same* structure, except that the top-level directory should be called `handin`, and be packaged up as `handin.zip`. You may edit the provided files that indicate that they should be edited, and/or add new ones as appropriate, but do not modify the overall directory structure. Before submitting, be sure to check that your project builds and tests as expected.

Please take care to include only files that constitute your actual submission: your source code and directly supporting materials (e.g., tests, build dependencies, any non-standard build/run instructions if relevant, etc.), but *not* obsolete/experimental versions of your code, backups (`*~`), autosave files (`#*#`), build directories (`.stack-work/`), and the like.

Content As always, your code should be appropriately commented. In particular, try to give brief informal specifications for any auxiliary "helper" functions you define, whether locally or globally. On the other hand, avoid trivial comments that just rephrase what the

code is already saying. Try to use a consistent indentation style, and avoid lines of over 80 characters.

You may import additional functionality from the GHC libraries. Do not use any packages that are not part of the Stack LTS-12.6 distribution. If you use any libraries that are not installed by default, be sure to mention them as dependencies in `package.yaml`.

Your code should give no warnings when compiled with `ghc(i) -W`; otherwise, add a comment explaining why any such warning is harmless or irrelevant in each particular instance. If some problem in your code prevents the whole file from compiling at all, be sure to comment out the offending part.

Report

In addition to the code, you must submit a short (normally 2–3 pages) report, covering the following two points:

- Document any (relevant) *design* and *implementation* choices you made. This includes, but is not limited to, answering any questions explicitly asked in the assignment text. Focus on high-level aspects and ideas, and motivate *why* you did something, not only *what* you did. It's rarely appropriate to do a detailed function-by-function code walk-through in the report; non-trivial remarks about how the functions work belong in the code as comments.
- Give a honest, justified *assessment* of the quality of your submitted code, and the degree to which it fulfills the requirements of the assignment (to the best of your understanding and knowledge). Be sure to clearly explain any known or suspected deficiencies.

  It is very important that you document on what your assessment is based, including the tests and/or proofs you did. Include your automated tests with your source submission (explaining how to run them, so we can reproduce your testing), and summarize the results in the report.

Your report submission should be a single PDF file named `report.pdf`, uploaded along with `handin.zip`. It should include a listing of your code (but not the already provided auxiliary files) as an appendix.

General

Detailed upload instructions, in particular regarding the logistics of group submissions, can be found on the Absalon submission page.