Assignment 1: A SubScript Interpreter

Based on a task from the 2015/16 final exam

Version of 12th September 2017

JavaScript is perhaps the most widely available programming language in the world: nearly every personal computer and handheld device, connected to the Internet, has some sort of JavaScript engine installed, as part of the web browser engine, or otherwise.

Perhaps to make JavaScript more comprehensible, a recurring proposal is to add array comprehensions to JavaScript. (These are similar list comprehensions in Haskell.) One implementation can be found in Firefox version 30, or greater[1].

This assignment is about implementing an interpreter for a conservative subset of Mozilla's JavaScript implementation, which we will call SubScript.

It is not part of this assignment to implement a parser for SubScript. Your task is to implement a SubScript *interpreter*, which given a SubScript abstract syntax tree either yields a value or an error. (This is specified more formally 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.

```haskell
module SubsAst where

data Expr = Number Int
          | String String
          | Array [Expr]
          | Undefined
          | TrueConst
          | FalseConst
          | Var Ident
          | Compr ArrayCompr
          | Call FunName [Expr]
          | Assign Ident Expr
```

[1]For more details on the Firefox implementation of JavaScript array comprehensions, see `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Array_comprehensions`.

```
          | 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
```

Mozilla-style JavaScript Array Comprehensions

You can use the Firefox Web Console Developer Tool[2] to get to a Mozilla JavaScript prompt. This is a simple way to play around with JavaScript array comprehensions to understand how they behave.

SubScript syntax is explained in-depth in appendix B. Since writing a SubScript parser is not part of this assignment, we proceed to explain the relationship between the above abstract syntax tree and what you can type in your Firefox Web Console Developer Tool by example.

Consider an array of numbers:

```
xs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This corresponds to the following `Expr`:

```
Assign "xs"
 (Array [Number 0, Number 1, Number 2, Number 3, Number 4,
         Number 5, Number 6, Number 7, Number 8, Number 9])
```

You can get an array of the squares of the numbers as follows:

```
[ for (x of xs) x * x ]
```

This corresponds to the following `Expr`:

```
Compr (ACFor ("x", Var "xs")
        (ACBody (Call "*" [Var "x", Var "x"])))
```

You can also filter the array according to a predicate, and get a, perhaps, smaller array. For instance, to get all the even numbers in an array of numbers:

```
[ for (x of xs) if (x % 2 === 0) x ]
```

This corresponds to the following `Expr`:

[2] `https://developer.mozilla.org/en/docs/Tools/Web_Console`

```
Compr (ACFor ("x", Var "xs")
        (ACIf (Call "===" [Call "%" [Var "x", Number 2], Number 0])
          (ACBody (Var "x"))))
```

You can also perform nested iterations, and generate larger arrays. For instance, to repeat an element (in this case) 100 times:

```
[ for (x of xs) for (y of xs) 'a' ]
```

This corresponds to the following `Expr`:

```
Compr (ACFor "x" (Var "xs")
        (ACFor "y" (Var "xs")
          (ACBody (String "a"))))
```

Or, to generate (in this case) 100 consecutive integers, starting at 1:

```
[ for (i of [0]) for (x of Array(5)) for (y of Array(20)) i = i + 1 ];
```

This corresponds to the following `Expr`:

```
Compr (ACFor "i" (Array [Number 0])
        (ACFor "x" (Call "Array" [Number 5])
          (ACFor "y" (Call "Array" [Number 20])
            (ACBody (Assign "i"
                      (Call "+" [Var "i", Number 1]))))))
```

The Interpreter

This assignment is about writing an interpreter for the SubScript subset of JavaScript. The intention is that the semantics of a SubScript program is mostly the same as if it would be as interpreted by a standard JavaScript interpreter up to a few simplifications with respect to type coercions, which are detailed in the following.

We recommend that you read through the whole question before you start implementing anything, and read the examples in the introduction and in Appendix A, which contains both the (rather more readable) source syntax and its representation as abstract syntax trees.

Semantics of SubScript

The semantics of most of SubScript should be straightforward, below some of the more murky points are elaborated.

- Variables can only be referred to after they have been initialized, either by an assignment, or by a `for` clause in a comprehension.

- The value of an assignment expression, `x = e`, is the value of the right-hand side, `e`.
- The comma operator, `e1, e2`, evaluates each of its operands (from left to right) and returns the value of the last operand, `e2`.
- In contrast to JavaScript, there are only limited type coercions in SubScript. Thus, it is illegal to, say, multiply an integer and an boolean (which is legal in JavaScript).

  Both arguments to arithmetic operators must be integers. The only exception to this rule is for addition, where it is possible to add two strings, or a string and a number (in any order); in the later case the number should be converted to a string first, as addition of strings means string concatenation.

  The === operator compares its arguments for structural equality, without any coercions. It accepts operands of any type, but comparison of, e.g., a string and a number will always yield `false`. On the other hand, the two arguments to the < operator must either both be integers or both be strings, where strings are compared using the usual lexicographic order.
- The built-in function `Array(e)`, where `e` must evaluate to a non-negative integer *n*, creates a array an array with *n* elements, all with the special value `undefined`.
- An array comprehension consists of zero or more `for-` clauses (of the form "`for (x of e)`") and/or `if`-clauses (of the form "`if (e)`"), followed by an expression to be evaluated. (In the concrete SubScript grammar, there is a requirement that the first clause must be a `for`-clause, but there is no particular reason to impose that restriction on the *abstract syntax* that the interpreter can handle.

  The result of an array comprehension is an array whose elements are generated as follows, inspecting the clauses from left to right:

  – A comprehension consisting of just a single expression `e`, generates exactly one element, namely the value of `e`.

  – For a comprehension starting with a `for`-clause "`for (x of ea)`", where `ea` evaluates to either an array or a string, the remainder of the comprehension after the clause is evaluated with `x` initialized to each element of `ea` in turn. (The elements of a string are its consitituent one-character substrings.)

  – A comprehension starting with an `if`-clause "`if (eb)`", where `eb` evaluates to `true`, generates the same elements as the remainder of the comprehension. (I.e., the clause has no effect in this case.)

  – A comprehension starting with an `if`-clause "`if (eb)`", where `eb` evaluates to `false`, generates no elements, and the remainder of the comprehension is ignored.

  If the expression in a `for-` or `if`-clause evalautes to a value of a different type than covered above, the result is a runtime error.

  The semantics of SubScript array comprehensions is this very similar to Haskell list comprehensions, with `for`-clauses corresponding to generators, and `if`-clauses corresponding to tests. Keep in mind, however, that all expressions in an array comprehension may havbe side effects (such as assigning to a variable, as exploited in the last example of the introduction), while their Haskell counterparts must be completely pure.

Note that the variables bound in `for`-clauses are only in scope in the remainder of the comprehension to the right. For example, in "`[ for (x of e1) for (y of e2) e3]`", the binding of `x` is visible in `e2` and `e3`, while the binding of `y` is only visible in `e3`. Assignments to a `for`-bound variable are legal, but only survive until the next iteration. After the entire comprehension has been evaluated, a `for`-bound variable reverts to its original value (if any) outside the comprehension.

- The result of succesful evaluation of a SubScript expression is a value which is either an integer, a boolean, the special value `undefined` (not to be confused with Haskell's `undefined` expression, which causes a runtime error), a string, or an array of values. We represent values by the following Haskell data type:

```
data Value = IntVal Int
           | UndefinedVal
           | TrueVal | FalseVal
           | StringVal String
           | ArrayVal [Value]
           deriving (Eq, Show)
```

  which should be declared in `SubsInterpreter.hs`.

If your interpreter encounters an error, it should terminate with an well-defined error result. That is, **not** by calling the built-in Haskell function `error`.

Your Task

The main objective of this question is that you should demonstrate that you know how to write an interpreter using monads for structuring your code. Thus you should structure your solution along the following lines, where you most likely also need a few extra helper functions:

(a) Define a module `SubsInterpreter` that exports a function the type `Value` and a function `runExpr`.

See the handed-out `SubsInterpreter.hs` for a *strongly recommended* skeleton for your solution. The handed-out skeleton code already has the exports set up correctly.

(b) During the interpretation of a SubScript program we need to keep track of a context for the expressions to be evaluated in. The context consists of two parts: (1) a variable environment mapping variable names to values; and (2) a read-only primitives environment mapping names of built-in functions and operators (primitives) to Haskell functions implementing their semantics. That is, we use the following types:

```
type Error = String
type Env = Map Ident Value
type Primitive = [Value] -> Either Error Value
```

```haskell
type PEnv = Map FunName Primitive
type Context = (Env, PEnv)
```

where Map is from the Data.Map library. These types are already declared in the skeleton SubsInterpreter.hs.

(c) We use the type SubsM for structuring our interpreter:

```haskell
newtype SubsM a = SubsM {runSubsM :: Context -> Either Error (a, Env)}
```

Make the SubsM type a Monad instance (and a Functor and Applicative instances as well).

(d) In the initial context, initialContext, we have an empty variable environment, and a primitives environment with seven entries. Finish the following implementation:

```haskell
initialContext :: Context
initialContext = (Map.empty, initialPEnv)
  where initialPEnv =
          Map.fromList [ ("===", undefined)
                       , ("<", undefined)
                       , ("+", undefined)
                       , ("*", undefined)
                       , ("-", undefined)
                       , ("%", undefined)
                       , ("Array", mkArray)
                       ]
```

Where the primitive mkArray, for example, is implemented by the following function:

```haskell
mkArray :: Primitive
mkArray [IntVal n] | n >= 0 = return $ ArrayVal (replicate n UndefinedVal)
mkArray _ = Left "Array() called with wrong number or type of arguments"
```

(e) Implement the following utility functions for working with the context:

```haskell
modifyEnv   :: (Env -> Env) -> SubsM ()
putVar      :: Ident -> Value -> SubsM ()
getVar      :: Ident -> SubsM Value
getFunction :: FunName -> SubsM Primitive
```

(f) Implement a function for evaluating expressions:

```haskell
evalExpr :: Expr -> SubsM Value
```

(g) Implement a function runExpr

```haskell
runExpr :: Expr -> Either Error Value
```

runExpr *e* evaluates expression *e* in the initial context, yielding either a runtime error, or the value of the expression. Note that if you also want to see the final value of one or more variables, you can package them up as part of the result using an explicit array constructor, as in,

```
xs = [1,2,3,4],
a = 0,
sxs = [for (x of xs) a = a + x],
[xs, a, xs]
```

Advice for your solution

Getting array comprehension right is the most difficult part of this assignment. Hence, if you have difficulties making your interpreter work for the full SubScript language, then start by making it work for the subset of the language with array comprehensions left out.

Then proceed by, for instance, allowing only one `for` clause in array comprehensions, or disallowing `if` clauses in array comprehensions, and so on.

If you make such restrictions make sure to clearly documenting them in your assessment, and explain why the disallowed language constructs cause you problems.

Also, make sure that you have tested your solution and that your testing is automated, so that we can run your tests and verify your test results.

A `Subs` program

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

```
$ runhaskell Subs.hs intro-ast.txt
```

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 Discuss in-how-far your solution supports array comprehensions.

   b Explain how your monad instance for `SubsM` works, and give *some* evidence that it satisfies the monad laws. (This could be in the form of a formal or semi-formal proof, some relevant test cases, or similar.)

   c 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, with the parameter `-W`, should not yield any errors or warnings for your code.

6. `hlint` should not yield any hints for your code.

Appendix A: Example SubScript programs

Appendix A.1: Source code for intro.js

The examples from the introduction for SubScript.

```
xs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
squares = [ for (x of xs) x * x ],
evens = [ for (x of xs) if (x % 2 === 0) x ],
many_a = [ for (x of xs) for (y of xs) 'a' ],
hundred = [ for (i of [0])
            for (x of Array(5))
            for (y of Array(20)) i = i + 1 ],
[xs, squares, evens, many_a, hundred]
```

Appendix A.2: Abstract syntax tree for intro.js

```
Comma (Assign "xs"
        (Array [Number 0, Number 1, Number 2, Number 3, Number 4,
                Number 5, Number 6, Number 7, Number 8, Number 9]))
 (Comma (Assign "squares"
          (Compr (ACFor "x" (Var "xs")
                   (ACBody (Call "*" [Var "x",Var "x"])))))
   (Comma (Assign "evens"
            (Compr (ACFor "x" (Var "xs")
                     (ACIf (Call "===" [Call "%" [Var "x", Number 2],
                                        Number 0])
                       (ACBody (Var "x"))))))
     (Comma (Assign "many_a"
              (Compr (ACFor "x" (Var "xs")
                       (ACFor "y" (Var "xs")
                         (ACBody (String "a"))))))
       (Comma (Assign "hundred"
                (Compr (ACFor "i" (Array [Number 0])
                         (ACFor "x" (Call "Array" [Number 5])
                           (ACFor "y" (Call "Array" [Number 20])
                             (ACBody (Assign "i"
                                       (Call "+" [Var "i", Number 1]))))))))
         (Array [Var "xs", Var "squares", Var "evens",
                 Var "many_a", Var "hundred"])))))
```

Appendix A.3: Source code for scope.js

Simple program that demonstrates that variables bound in array comprehensions can shadow those declared before, in this example the variable x, and that those bindings are restored afterwords. Thus, in this example we end up with `x` bound to the value `42`, and the variable `y` bound to an array with three elements, each a one-character string.

```
x = 42,
y = [for (x of 'abc') x],
[x, y]
```

Appendix A.4: Abstract syntax tree for `scope.js`

```
Comma (Assign "x" (Number 42))
 (Comma (Assign "y" (Compr (ACFor "x" (String "abc")
                             (ACBody (Var "x")))))
   (Array [Var "x", Var "y"]))
```

Appendix B: Grammar

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Expr}\ \texttt{','}\ \mathit{Expr} \\
 & | & \mathit{Expr1} \\
\mathit{Expr1} & ::= & \mathit{Number} \\
 & | & \mathit{String} \\
 & | & \texttt{'true'} \\
 & | & \texttt{'false'} \\
 & | & \texttt{'undefined'} \\
 & | & \mathit{Ident} \\
 & | & \mathit{Expr1}\ \texttt{'+'}\ \mathit{Expr1} \\
 & | & \mathit{Expr1}\ \texttt{'-'}\ \mathit{Expr1} \\
 & | & \mathit{Expr1}\ \texttt{'*'}\ \mathit{Expr1} \\
 & | & \mathit{Expr1}\ \texttt{'\%'}\ \mathit{Expr1} \\
 & | & \mathit{Expr1}\ \texttt{'<'}\ \mathit{Expr1} \\
 & | & \mathit{Expr1}\ \texttt{'==='}\ \mathit{Expr1} \\
 & | & \mathit{Ident}\ \texttt{'='}\ \mathit{Expr1} \\
 & | & \mathit{Ident}\ \texttt{'('}\ \mathit{Exprs}\ \texttt{')'} \\
 & | & \texttt{'['}\ \mathit{Exprs}\ \texttt{']'} \\
 & | & \texttt{'['}\ \mathit{ArrayFor}\ \texttt{']'} \\
 & | & \texttt{'('}\ \mathit{Expr}\ \texttt{')'} \\
\mathit{Exprs} & ::= & \epsilon \\
 & | & \mathit{Expr1}\ \ \mathit{CommaExprs} \\
\mathit{CommaExprs} & ::= & \epsilon \\
 & | & \texttt{','}\ \mathit{Expr1}\ \ \mathit{CommaExprs} \\
\mathit{ArrayFor} & ::= & \texttt{'for'}\ \texttt{'('}\ \mathit{Ident}\ \texttt{'of'}\ \mathit{Expr1}\ \texttt{')'}\ \mathit{ArrayCompr} \\
\mathit{ArrayIf} & ::= & \texttt{'if'}\ \texttt{'('}\ \mathit{Expr1}\ \texttt{')'}\ \mathit{ArrayCompr} \\
\mathit{ArrayCompr} & ::= & \mathit{Expr1} \\
 & | & \mathit{ArrayFor} \\
 & | & \mathit{ArrayIf}
\end{array}
$$

Keywords and special symbols are written between single quotes, and ϵ represents an empty string.

Note that this grammar does not yet fully specify the syntax of SubScript programs:

- It is *incomplete*, in that we have not given any specification of the nonterminals *Ident*, *Number*, and *String*, which represent, respectively, SubScript variable names, numeric literals, and string literals. To complete the grammar, we would need to specifiy, e.g., exactly what characters are allowed in variable names.

- It is *ambiguous*, e.g., in that it does not specify how expressions with multiple operators, such as `2 + 3 * x`, are to be read: as `(2 + 3) * x`, or as `2 + (3 * x)`? To resolve such ambiguities, we need to also stipulate the precedence and associativity conventions to be used.

Both of those questions have to do with *parsing*, which we will cover later in the course. For the purpose of this assignment, we will assume that input programs have already been parsed into *abstract syntax trees*, and are thus represented as values from a collection of suitable Haskell algebraic datatypes.