Assignment 1: A SubScript Interpreter

Based on a task from the 2015/16 final exam

Version 1.00

Due: Wednesday, September 19 at 20:00

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 would be similar to list comprehensions in Haskell. One implementation can be found in Firefox versions 30–58[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, returns either a value or an error message. (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.

```
module SubsAst where

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

```
          | 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
```

Mozilla-style JavaScript Array Comprehensions

If you have (or can install) a suitable Firefox version, you can use its Web Console Developer Tool[2] to get to a Mozilla JavaScript prompt. Alternatively, you may be able to install just the stand-alone SpiderMonkey JavaScript engine.[3] These give you a simple way to play around with "real" JavaScript array comprehensions, but are *not* a requirement for this assignment: the expected semantics is also fully described in the following.

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 would write in JavaScript 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:

[2] https://developer.mozilla.org/en/docs/Tools/Web_Console
[3] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Releases/52

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

This corresponds to the following `Expr`:

```
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 latter case, the number is 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 constituent 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 evaluates to a value of a different type than covered above, the result is a runtime error.

The semantics of SubScript array comprehensions is thus 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 have 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 successfully evaluating 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:

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

  which should be declared in module `SubsInterpreter`.

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 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
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:

```
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). The definitions of `return` and `>>=` should reflect that the variable environment gets modified during the computation, while the primitives environment stays the same throughout. The `fail` function should use the error-reporting facilities of the monad, not the `error` function.

In the *report*, you should briefly 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.)

Hint: try to express the monad operations of `SubsM` in terms of the monad operations of `Either Error`; this will make your code both more compact and easier to reason about.

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

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

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

```
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:

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

(f) Implement a function for evaluating expressions:

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

(g) Implement a function `runExpr`

```
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,
ys = [for (x of xs) a = a + x],
[xs, a, ys]
```

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 document 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.

An interpreter program

We also hand out a file `Main.hs` with a stand-alone interpreter that reads a SubScript program from a file and evaluates it. Once you have something that begins to look like an interpreter, you can test it as follows:

```
$ runhaskell Main.hs ../examples/intro-ast.txt
```

What to hand in

Code

Form Your main code should be placed in the file `SubsInterpreter.hs` (and, if relevant, in other modules from which `SubsInterpeter` imports). It should only implement the requested functionality. Any tests or examples should be put in separate files.

We have provided a skeleton `SubsInterpreter.hs` with stub definitions for all the functions you are expected to define. It is very important that you not change the types of the exported functions, as this will make your code incompatible with our testing framework. Do not remove the bindings for any functions you do not implement; just leave them as `undefined`.

The definitions for this assignment (e.g. type `Expr`) are available in file `SubsAst.hs`. You should only import this module, and not directly copy its contents into `SubsInterpreter.hs`. And again, do not modify anything in `SubsAst`.

Important: 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, 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 `src.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.

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

```
          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
```

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.