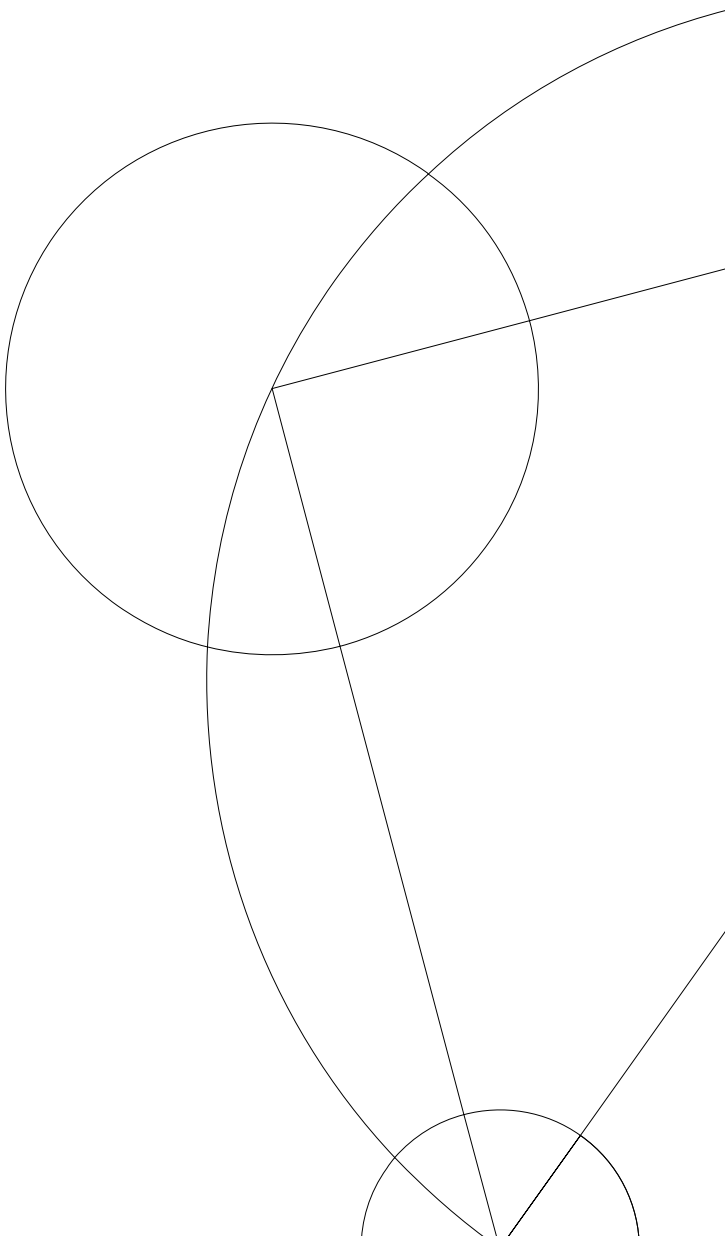
UNIVERSITY OF COPENHAGEN

AP Assignment 1 - SubScript Interpreter

DIKU University of Copenhagen 2017

Jan Sokol, Denis Trebula
{kgj360,jmp640}@alumni.ku.dk

September 28, 2017



1 Subscript Interpreter

The goal of this assigment is about implementing an interpreter for conserative subset of Mozilla's JavaScript implementation, which we will call SubScript.

The interpreter can be run in two options. Option a, using helper functions provided in assessement, by running `stack runhaskell Subs.hs AST.txt` in commandline, where AST.txt is an plain text file containing Abstract syntax tree of some SubScript code. Option b, by loading Tests.hs file into GHCI and running mainTxtTest. which also uses file intro-ast.txt.

1.1 Implementation

We can seperate our implementation into two main sections.

1.1.1 Initial types and Monad

SubM monad we had to implement is quite similar to Haskell's State monad [1] so we took some inspiration there (we also keep environments that we're passing for future evaluations)

For construction of SubsM instance of type a we have to provide function of type:

```
Context -> Either Error (a, Env)
```

If we want to deconstruct the instance, type constuctor contains function runSubsM.

1.1.2 Monad Laws

If something is called monad, it has to follow 3 rules [3]:

- Left identity,
- Right identity,
- Associativity laws.

Left identity `(return a >>= f === f a)` This law states that `(return a) bind f == f a` or Create a monad with 'a' and bind it with 'f' is the same as calling 'f(a)'. We can prove this law in: [2]

```
return a >>= f -- def. of return
= SubsM (\(env, pEnv) -> Right (a, env)) >>= f -- def. of >>=
= SubsM (\(env, pEnv) -> runSubsM (f a) (env, pEnv)) -- def. of runSubsM
= f a
```

Right identity `(m >>= return === m)` This law states that `m bind return == m` or Binding return on a monad returns the same monad. Here we have two cases:

```
runSubsM m c = Left e
(SubsM x) >>= return -- def. of >>=
= SubsM (\(env, pEnv) -> Left e) -- def. of SubsM x
= (SubsM x)
```

And second case:

```
(SubsM x) >>= return -- def. of >>=
runSubsM (return x) (env,pEnv) -- def. of return
= return x
= SubsM x
```

Associativity law This law states that `m bind f bind g == m bind (i -> f(i) bind g)` or Binding f to a monad and then binding g is the same as binding a function that produces a monad using f bidning g to the result to the same monad. [2]

Proving this law formally is rather complex, so we provided an example in our tests. Function to test Associativity law which takes expression as argument, looks like:

```
testCase "Monad Associativity Law Test 1" $
            (runSubsM (do asoc <- do { x <- return Undefined;
                                       ex x
                                     }
                          mon asoc
                      ) initialContext @?= runSubsM (do x <- return Undefined
                                                        do { asoc <- ex x;
                                                             mon asoc
                                                           }
                                                    ) initialContext)
```

Where "Undefined" represents an expression.

Implemented functions In this paragraph we will show our implementation of primitive functions which take list of values and return an Either type. If expected types are not matches, functions will yield errors.

Primitive functions functions are functions that take list of values and return an `Either` type. If expected types are not matched, fuctions will yield errors.

- `primitiveEq` is a strictly equal function, so the arguments has to be structurally equal. For example comparing int with string will return `False`
- `primitiveLt` makes comparison between 2 ints or 2 strings and evaluates as true if the one is less than the other.
- `primitiveAnd` makes addition between 2 ints, strings or combination of them and vice versa. If one of the arguments is string, function will return a String
- `primitiveMul` makes multiplication between 2 integers.
- `primitiveSub` represents mathematical function minus.
- `primitiveMod` uses modulo function on two arguments and return the result
- `mkArray` was given forehand in the assigment.

Example of primitive function:

```
primitiveMul :: Primitive
primitiveMul [IntVal num, IntVal num' ] = return (IntVal  (num * num'))
primitiveMul _ = Left " `*` arguments must be 2 integers"
```

modifyEnv This is a very simple one line function where we just update given enviroment.

putVar This function uses identificator and value as arguments and adds them together in a Map.

getVar This function is basically just a getter function for value, we use function lookup in a map for the name of the variable.

getFunction This function is also a getter function which looks up a given function name as an argument in our list of primitive functions. We use this function in evalExpr(Call) function so we can know which one of our primitive functions to call.

Example of `getFunction` from our code:

```
getFunction :: FunName -> SubsM Primitive
getFunction funName = SubsM (\(env, penv) -> case Map.lookup funName penv of
                                Just p -> Right (p, env)
                                Nothing -> Left ("unbound function" ++
                                                 "used but not defined"
                                                 ++ funName))
```

evalExpr This function is basically bread and butter of our interpreter. Its main purpose is to evaluate given expression and return a value based on chosen process. We have created evaluation of Number, String, Array, Undefined, TrueConst, FalseConst, Var, Call, Assign, Comma, Comprehension.

Example of evalExpr function:

```
evalExpr (Comma expr1 expr2) = evalExpr expr1  >> evalExpr expr2
```

runExpr runExpr evaluate expression e in the initial context, yielding either a runtime error or the value of the expression.

```
runExpr :: Expr -> Either Error Value
runExpr expression = case runSubsM (evalExpr expression) initialContext of
   Left err -> Left err
   Right (valOfExpr, _) -> Right valOfExpr
}
```

1.1.3 List Comperehention

We split array comprehention into 3 cases:

- ACBody – we only evaluate expression inside.
- ACFor – takes 3 parameters: indentifier, expression and array compehention. For evaluating ACFor we use helper function. In evalACFor we consider 2 main cases – if we are iterating over string, we change it into array of chars. If we iterate over array, we run evalACFor recursively over each item in array. Function includes code for flattening arrays. Other cases will evaluate as error.

- ACIf – takes two parameters. First one is expression and second another nested array comprehention. If the expression gets evaluated to tue, then Comprehention also evaluates. Otherwise function returns empty array.

Code for ACIf is shown below:

```
evalExpr (Compr (ACIf expr comp)) = do
  trueOrFalse <- evalExpr expr
  if trueOrFalse == TrueVal
    then evalExpr (Compr comp)
    else if trueOrFalse == FalseVal
      then return (ArrayVal[])
      else SubsM (const (Left "ACIf must be supplied with a boolean condition"))
```

We acknowledge that in order to return a succesfull value we need to save Ident value for ACFor and restore the old one after evaluating (x is only local variable here.)

```
Comma (Assign "x" (Number 1))
      (Comma (Compr (ACFor "x"
                           (Array [Number 2,Number 3])
                           (ACBody (Var "x"))))
             (Var "x"))
```

1.2 Testing - How to run code and reproduce test reults

We find it important to keep tests in separate file. We keep all the tests in file Test.hs and HunitTests.hs. We have implemented over 16 automated tests and 21 Hunit test cases using tasty (testing) framework and Hunit. We created tests to check simple primitive functions but also more complicated ones like array comprehensions for ACFor, ACIf, ACbody. We also used main and nice functions from Subs.hs file to create our own test in Tests.hs which allows us to test file "intro-ast.txt" with all of our testing functions together.

We have also implemented 6 Hunit tests for Monad laws, 2 for each law, to provide evidence that our choice of Monad indeed satisfies the monad laws.

Example:

```
testCase "Monad Right Identity Law Test 1" $
          (runSubsM (do right <- ex Undefined
                        return right
                        ) initialContext @?= runSubsM (do ex Undefined
                                                          ) initialContext ),
```

1. In order to reproduce our results from automated testing all you need is to load Tests.hs within GHCI and type mainTxtTest which uses mentioned "intro-ast.txt" from Main or you can type testAll to get output from every test case we implemented.

2. In order to reproduce our Hunit testing all you need is to type:

   ```
   stack exec ghci -- -W UnitTests.hs
   ```

Take in mind that in order to successfully compile the file you must have tasty framework and its plugin Hunit installed. To do so, you need to type:

```
stack install tasty
stack install tasty-hunit
```

Then you simply write main and you will reproduce our results.

Example of simple and more complicated tests:

```
correctLessTest :: Bool
correctLessTest = (runExpr(Call "<" [Number 4, Number 8]) == Right TrueVal)

correctAssignTestInt :: Bool
correctAssignTestInt = case runExpr (Comma (Assign "f" (Number 8))
                        (Comma (Assign "d" (Number 4))
                        (Var "f"))) of
                          Right v -> v == IntVal 8
                          Left _ -> False
```

1.3 Assessment

We were able to develop a solid version of SubScript interpreter. We were able to finish all points of the assigment. We have implemented also checkers for error values in our functions.

1.3.1 Support for our claims

To support our claims we have implemented 37 test cases(16 automated tests, 21 Hunit tests) covering Monad laws, AST trees given in the assesement directory and tests of runExpr function. We used tests for simple primitive functions but also for more complicated array comprehensions and monad laws.

1.3.2 Summary of the Code Quality

Based on the tests, we were capable of solving the task and we believe that our solution is adequate and likely to be correct. Despite the fact we tested 37 test cases, we recognize that we couldnt test every possible scenario that could happen. Additionally we used OnlineTA test which our solution passed without failure, warning or hint and showed us that Interpreter works as expected.

The quality of code and test cases could be improved, for example using Hunit to generate random expressions, or by using more functions from haskell standad library instead of implementing our own. Because of lack of time we were not able to improve code quality further.

1.4 Resources

[1] https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State
[2] https://hkupty.github.io/2016/Understanding-the-math-behind-fp-monads/
[3] https://wiki.haskell.org/Monad_laws