Advanced Programming 2017
Assignment 1

Tom Jager
dgr418@alumni.ku.dk

Tobias Ludwig
fqj315@alumni.ku.dk

September 20, 2017

1 SubsM ...

`SubsM` is a datatype that when given an argument, it returns a function that takes a context and returns either an error or a tuple containing an item and a variable environment.

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

It is used to evaluate expressions in the Subscript language in a given context and return either errors or the value returned from the expression with an updated variable environment. As such it can be seen as a type of state transformer. For use in the evaluator, `SubsM` must be implemented as an instance of a Functor and as a Monad, it also has been implemented as an Applicative functor but is not used as such yet.

1.1 Functor

```
instance Functor SubsM where
  fmap f fra = SubsM (\c -> case runSubsM fra c of
    (Left e)          -> Left e
    (Right (a,env)) -> Right (f a,env))
```

The functor instance is a demonstration on how the `SubsM` functor should behave under the fmap function. As such when fmap is provided with a function `f :: a -> b` and a functor `fra :: SubsM a` , fmap extracts the item of type `a` from the functor and applies it to `f`, re-wrapping the result into the functor. By implementing it in this fashion, all side-effects from evaluating the `SubsM a` are kept.

1.2 Applicative Functor

```
instance Applicative SubsM where
  pure a = SubsM (\c -> Right (a, fst c))
  ff <*> fa = SubsM (\c0 -> case runSubsM ff c0 of
    (Left e)          -> Left e
    (Right (f, env)) -> case runSubsM fa (env,snd c0) of
      (Left e')          -> Left e'
      (Right (a, env')) -> Right (f a, env'))
```

The Applicative functor instance demonstrates how `SubsM` behaves under the applicative operation. `Pure a` is used to create a functor of type a in a fashion that is side-effect free. `<*> :: SubsM (a -> b) -> SubsM a -> SubsM (b)` is an operation that unwraps a function and a value from the functor, applies them to each other and then re-wraps the result. It tracks all the side-effects in the process

1.3 Monad

```
instance Monad SubsM where
  return x = SubsM (\c -> Right (x, fst c))
  m >>= f  = SubsM (\c0 -> case runSubsM m c0 of
     (Left e)          -> Left e
     (Right (x,env)) -> case runSubsM (f x) (env,snd c0) of
       (Left e')           -> Left e'
       (Right (x',env')) -> Right (x',env'))
  fail s = s
```

The most powerful instance of `SubsM` is defining it as a Monad. This involves declaring its behavior over three functions, `return`, `(>>=)` and `fail`. `return x` is identical to `pure a` in that it takes a value and returns it as a SubsM monad, side-effect free. `(>>=) :: SubsM a -> (a -> SubsM b) -> SubsM b` takes a `SubsM a`, extracts the `a` value and returns the result of applying the function `f` to it. This maintains all side-effects and error handling. Finally `fail s` occurs when `f` does not receive the Monad's type as an argument and returns an error string.

1.4 Monad Laws

There are 3 Monad Laws which every Monad must obey. These are the Left Identity, the Right Identity and the Associativity Laws. I will demonstrate the first two with proofs and the last by example.

1.4.1 1. Left Identity (`return a >>= f === f a`)

```
return a >>= f
—{By definition of return}
= SubsM (\c -> Right (a, fst c)) >>= f
—{By definition of >>= and where runSubsM (return a) c = (a,fst c)}
= SubsM (\c -> runSubsM (f a) (fst c,snd c))
—{By definition of runSubsM}
= f a
```

1.4.2 2. Right Identity (`m >>= return === m`)

```
Case 1: runSubsM m c = Left e
(SubsM x) >>= return
—{By definition of >>= }
= SubsM (\c -> Left e)
—{By definition of (SubsM x)}
= (SubsM x)

Case 2: runSubsM m c = Right (x,env)
(SubsM x) >>= return
```

```
—{By definition of >>=}
runSubsM (return x) (env,snd c)
—{By definition of return}
= return x
—{By definition of return}
= SubsM x
```

1.4.3 3. Associativity

As associativity is a complex law to prove, I have provided and tested an example. Associativity follows with monads where `(m >>= f) >>= g === m >>= (\x -> f x >>= g)`. The following code provides an example of the two cases.

```
assocEx1 :: Int -> SubsM Int
assocEx1 n = do
  y <- do
    x <- succSubsM n
    squareSubsM x
  mult10SubsM y

assocEx2 :: Int -> SubsM Int
assocEx2 n = do
  x <- succSubsM n
  y <- squareSubsM x
  mult10SubsM y
```

As quickCheck passes on a function checking that these two examples are equivalent. We can assume the associativity law holds. This quickCheck can be performed by calling `assocTest` from SubsTests.

2 Implementing the Primitives

Primitives are functions from a list of values to an `Either` type. For each operator we generate Errors if the arguments supplied do not match the expected number or types.

strictEquals is strict because it ensures that same types are compared and no coercion works here. We do this by checking the types of the arguments and their values. A special case is the `UndefinedVal` which we consider to be equal to nothing, not even another `UndefinedVal`, thus returning always `FalseVal`.

lessThan works for two `IntVal`s or two `StringVal`s (here lexicographically).

plus is polymorphic for `IntVal`s and `StringVal`s like described.

mult and sub only work on `IntVal`s.

mod catches the special case of "mod by zero" which is not defined, thus returning an error.

3 Utility functions for working with the context

modifyEnv takes an (abstract) function which is applied to the `Env` of the current context and returns a new one. The result is wrapped in a context where we do not need the value of the expression, setting it to `()`.

```
modifyEnv :: (Env -> Env) -> SubsM ()
modifyEnv f = SubsM (\c -> Right ((),f (fst c)))
```

putVar can now use `modifyEnv` by just calling it with the `Map.insert` function passing also the name and value of the variable to be inserted. This works because `Env` is of type `Map Ident Value`. The modiefied environment is returned.

```
putVar :: Ident -> Value -> SubsM ()
putVar name val = modifyEnv (Map.insert name val)
```

getVar also uses `modifyEnv` only with the `Map.lookup` function providing the name and the current environment. If the variable is not in there, we report an error, if it is, just the value.

```
getVar :: Ident -> SubsM Value
getVar name = SubsM (\c -> case Map.lookup name (fst c) of
  Nothing -> Left "Variable not initialised"
  (Just x)  -> Right (x,fst c))
```

getFunction also performs a lookup but in the second map containing the mapping from the Subscript operators to the Primitive functions.

```
getFunction :: FunName -> SubsM Primitive
getFunction name = SubsM (\c -> case Map.lookup name (snd c) of
  Nothing  -> Left "Function name not initialised"
  (Just x) -> Right (x,fst c))
```

4 Implementation of evalExpr and runExpr

4.1 evalExpr

`evalExpr :: Expr -> SubsM Value` is a function that takes an expression from the Abstract Syntax Tree and provides a value for it. It returns a state transformer that will evaluate the expression on a given context.

The first five expressions are simple to evaluate as they do not change the context but simply return their corresponding value.

```
evalExpr (Number x)  = SubsM (\c -> (Right ((IntVal x),fst c)))
evalExpr (String s)  = SubsM (\c -> (Right ((StringVal s),fst c)))
evalExpr Undefined   = SubsM (\c -> (Right (UndefinedVal,fst c)))
evalExpr TrueConst   = SubsM (\c -> (Right (TrueVal,fst c)))
evalExpr FalseConst  = SubsM (\c -> (Right (FalseVal,fst c)))
```

The `Array` value requires the use of the `mapM` function from the Control.Monad package. This function applies the `evalExpr` function to each expression in the array and returns a `SubsM [Value]` with all the evaluated values and the side effects. This is bound to `return (ArrayVal vals)` so the list of evaluated values is returned as an array value and all the side effects are preserved.

```
evalExpr (Array exprs) = do
  vals <- (mapM evalExpr exprs)
  return (ArrayVal vals)
```

The expression `Var Ident` is evaluated by simply running `getVar` with the identifier and returning the resulting `SubsM`.

```
evalExpr (Var name)  = getVar name
```

Evaluating array comprehensions is slightly more complex. In order to ensure scope rules are obeyed, the resulting monad contains the evaluated value from the comprehension but an un-altered context. This is not a perfect scope solution as it means that anything inside a comprehension does not affect the context, not just assignments in the for loop but due to time constraints this was not fixed.

```
evalExpr (Compr aComp) = SubsM (\c -> case runSubsM m c of
  Left e     -> (Left e)
  Right res -> (Right (fst res, fst c))) where
    m = do
      val <- evalCompr aComp
      return val
```

`evalCompr` is a function that evaluates an array comprehension. If the comprehension is simply an `ACBody` it performs `evalExpr` on the expression inside. For comprehensions are separated into loops across strings and loops across arrays but both perform in a similar manner. `evalCompr` uses `mapM` to perform the `iter` monadic function over each element of the array returning a `SubsM` of a list of values. This is then returned as a `SubsM ArrayVal`. The iterated function takes an element of an array or string and uses `putVar` to assign the given identifier to it in the context. The rest of the comprehension is evaluated under that context and the result is returned. In the case of neither a string nor an array being iterated through, the fail function will be called and return the default Haskell error message of incorrect type.

```
evalCompr :: ArrayCompr -> SubsM Value
evalCompr (ACBody expr)                        = evalExpr expr

evalCompr (ACFor name (String string) comp) = do
  vals <- mapM iterS string
  return (ArrayVal vals) where
    iterS x = do
        val <- evalExpr (String [x])
        putVar name val
        evalCompr comp
evalCompr (ACFor name expr comp) = do
  (ArrayVal array) <- evalExpr expr
  vals <- mapM iterAr array
  return (ArrayVal vals) where
    iterAr x = do
```

```
    putVar name x
    evalCompr comp
```

If comprehensions evaluate the expression given with the statement and binds the value of it to the variable `bool`. `checkBool` is then used to determine if the value is True, False or not a boolean value. If the value is true then the rest of the comprehension is evaluated and returned. If the value is false then an empty array is returned. This is not in the semantics of the Subscript language as it should simply not evaluate the array but we were unable to come up with a satisfactory solution in time. If the value is neither true nor false it produces the error message "If must be supplied with a boolean expression".

```
evalCompr (ACIf expr comp) = do
  bool <- evalExpr expr
  if checkBool bool == 1
    then evalCompr comp
    else if checkBool bool == 2
      then return (ArrayVal [])
      else SubsM (const (Left "If must be supplied with a boolean expression"))

checkBool :: Value -> Int
checkBool TrueVal  = 1
checkBool FalseVal = 2
checkBool _        = 3
```

The Call function is evaluated as follows. First the argument expressions are evaluated as a `SubsM [Value]` using `mapM`. The supplied identifier is used with `getFunction` to obtain a `SubsM Primitive` which is then run on the updated context obtained from evaluating the arguments. This reveals the correct primitive function which is applied to the list of argument values revealed earlier. The result of this is, whether it be an error or a value is then returned with the correct variable environment.

```
evalExpr (Call name exprs) = SubsM (\c -> case runSubsM (mapM evalExpr exprs) c of
  Left e           -> Left e
  Right (vals,env) -> case runSubsM (getFunction name) (env,snd c) of
    Left e           -> Left e
    Right (prim,env') -> case prim vals of
      Left e    -> Left e
      Right val -> Right (val,env'))
```

The Assign instruction simply binds the result of evaluating the given expression to a variable called `val`. `putVar` is then used to update the current context appropriately, return is used to produce the resultant `SubsM Value` with `val` as the value.

```
evalExpr (Assign name expr) = do
   val <- (evalExpr expr)
   putVar name val
   return val
```

Finally the Comma expression which evaluates one expression, followed by another. It does this by evaluating the first expression and binding the resultant value to _as it is unused. Next the second expression is performed and its value is the one returned.

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

```
  _ <- evalExpr expr1
  evalExpr expr2
```

4.2 runExpr

`runExpr` is a function that receives as input an expression or a program in the Subscript language. It evaluates this program over the initial context which is an empty variable environment and a procedural environment filled with all the primitive operations. If an error occurs during evaluation then the error is returned and displayed, otherwise the value evaluated from the entire expression is returned.

```
runExpr :: Expr -> Either Error Value
runExpr expr = case runSubsM (evalExpr expr) initialContext of
  Left e          -> (Left e)
  Right res -> (Right (fst res))
```

5 Testing

We mainly do unit testing for the operations, testing if expressions are interpreted to the right results and if the context is preserved. These come in addition to the tests that were given in the `*-ast.txt` files but are more specific. Property based testing seems to be not very applicable to this task because we have to create very specific constructions.

5.1 Expressions (context-independent)

The easiest thing to test are expressions like `Call "+" [(Number i1), (Number i2)]` because they do not change the context. To do so we wrote unit tests for the + and the $*$ operators which can be called with two integers and return a bool if the result of the interpreted expression was correct. The same we did for string concattenation. Automating these procedures is not really useful because arithmetic should work for all numbers if it works for some.

5.2 Assignments (context-dependent)

Testing the persistence of assigned variables is done also in form of unit tests (automated random assignments do not make sense). We test this for Integers, Strings and Arrays by assigning a initial value to it which we later read out by just stating `Var "var"`. To ensure we do not confuse the readout value with the value actually returned by the assignment itself we include another expression in between assign and read.

5.3 Array Comprehensions + Scope

Lastly we have some functions for testing array comprehensions. We tested the combinations of a single `ACFor`, a `ACFor` with `ACIf` and multiple `ACFor`s. They all work like expected.

The only test still failing is a function where we test whether comprehensions protect/maintain their own variables. For that purpose we assign an outer `Assign "x" (Number 5)` followed by a very simple Comprehension assigning `ACFor "x" (Array [Number 1])`. When we then test for the value of x after the Comprehension we get 1 instead of 5.

6 Conclusion

This version of the SubScript interpreter is a competent one. Error values are detected and reported but in the vast majority of correct expressions, correct values are evaluated. Whilst the scope properties for array comprehensions are not fully finished and falsely evaluating if comprehensions produce incorrect results, the rest of the interpreter works correctly.