

# A SubScript Interpreter

## Assignment 1

*Kai Arne S. Myklebust, Silvan Adrian*

Handed in: October 3, 2018



## Contents

<b>1 Design/Implementation</b>	<b>1</b>
<b>2 Code Assessment</b>	<b>2</b>
<b>A Code Listing</b>	<b>2</b>

## 1 Design/Implementation

Overall in this assignment our goal was to not use helper functions, where not specifically needed. This to make readability easier and making the code less complex since many of our helper functions from the previous assignment were unnecessary. We started by making a helper function for each arithmetic operation from "initial context". This was to get the first and second element from the list. After a while we got really annoyed doing that and found out that you can just get the first and second element by changing to [] list-brackets. We used head and tail in equality, but onlineTA says that we should not use them. We check for empty lists and a list of different length, so we check for possible errors which we found while testing from head and tail. In evalExpr we used the do notation to make it more readable when we have multiple actions in the same statement.

**EDIT RESUBMISSION:** We proof that our monad instance for SubsM satisfies the monad laws by testing each of the three monad laws individually. For the second Law we wrote a Test which we can't test for equality therefore the test is commented out (see comment in Test.hs) but we were able get the same result back by running it in ghci.

- `return v >>= f == f v`
- `m >>= (\a -> return a) == m`
- `(m >>= f) >>= g == m >>= (\a -> (f a >>= g))`

We worked our way through the monads, by doing many tests and figuring out what return values were needed. In addition we read a lot of monad introductions and articles and used more than 20 hours just to partly understand this assignment. Also we got help in another TA class. In the monad return we knew that we needed the right value of SubsM and worked our way from there. In fail we knew that we needed the left side value (Error). For the bind we used the slides from the lecture and the haskell wiki to work our way forward. Underway we did testing to see that we got the right return values.

## 2 Code Assessment

According to our own tests and onlineTa, everything except array compression works. Array compression was the hardest part and is only partly working. ACFor only works for arrays with numbers. If you have a String it sees the string as only one element and not multiple characters. ACFor does not work with nested for's. We weren't able to make nested for's working. We use putVar and know that it works. So the ACFor can see the variable, but one problem is that only the body should see the variable but now the whole ACFor sees the new variable. In ACIf we have the problem when the if clause evaluates to false it has to return a 'Value', but the assignment says it should return nothing. IF ACIf is inside a ACFor our solution does not work, but if there's a single ACIf then it works.

We ran our own tests to show these failures. These can be run by 'stack test' (the 6 out of 86 tests fail, which we also described in the assessment above).

One place where our test cases were able to help us find errors was in equality and having arrays of different lengths. We fixed it by checking for empty arrays and for different array lengths then we return a FalseVal.

## A Code Listing

---

```

1  module SubsInterpreter
2      (
3          Value(..)
4      , runExpr
5      , equality

```

```

6      , smallerThen
7      , add
8      , mul
9      , sub
10     , modulo
11     , mkArray
12     , SubsM
13     -- You may include additional exports here, if you want to
14     -- write unit tests for them.
15 )
16 where
17
18 import SubsAst
19
20 -- You might need the following imports
21 import Control.Monad
22 import qualified Data.Map as Map
23 import Data.Map (Map)
24
25
26 -- | A value is either an integer, the special constant
27   ↪ undefined,
28 -- true, false, a string, or an array of values.
29 -- Expressions are evaluated to values.
30 data Value = IntVal Int
31            | UndefinedVal
32            | TrueVal | FalseVal
33            | StringVal String
34            | ArrayVal [Value]
35            deriving (Eq, Show)
36
37 type Error = String
38 type Env = Map Ident Value
39 type Primitive = [Value] -> Either Error Value
40 type PEnv = Map FunName Primitive
41 type Context = (Env, PEnv)
42
43 initialContext :: Context
44 initialContext = (Map.empty, initialPEnv)
45   where initialPEnv =
46         Map.fromList [ ("==", equality)
47                       , ("<", smallerThen)
48                       , ("+", add)

```

```

49         , ("*", mul)
50         , ("-", sub)
51         , ("%", modulo)
52         , ("Array", mkArray)
53     ]
54
55 newtype SubsM a = SubsM {runSubsM :: Context -> Either Error (a,
56     ↳ Env)}
57
58 instance Monad SubsM where
59     return x = SubsM $ \ (e, _) -> Right (x, e)
60     m >>= f = SubsM $ \ c@(_, p) -> runSubsM m c >>= \ (x, e') ->
61         ↳ runSubsM (f x) (e', p)
62     fail s = SubsM $ \ _ -> Left s
63
64 -- You may modify these if you want, but it shouldn't be
65 ↳ necessary
66
67 instance Functor SubsM where
68     fmap = liftM
69
70 instance Applicative SubsM where
71     pure = return
72     (<*>) = ap
73
74 equality :: Primitive
75 equality [IntVal a, IntVal b] = if (a == b) then Right TrueVal
76     ↳ else Right FalseVal
77 equality [UndefinedVal, UndefinedVal] = Right TrueVal
78 equality [StringVal a, StringVal b] = if a == b then Right
79     ↳ TrueVal else Right FalseVal
80 equality [TrueVal, TrueVal] = Right TrueVal
81 equality [FalseVal, FalseVal] = Right TrueVal
82 equality [ArrayVal [], ArrayVal []] = Right TrueVal
83 equality [ArrayVal [], ArrayVal _] = Right FalseVal
84 equality [ArrayVal _, ArrayVal []] = Right FalseVal
85 equality [ArrayVal a, ArrayVal b] = if head a == head b
86     then equality [ArrayVal (tail a), ArrayVal (tail b)]
87     else Right FalseVal
88 equality [_ , _] = Right FalseVal
89 equality _ = Left "Wrong number of arguments"
90
91 smallerThen :: Primitive
92 smallerThen [IntVal a, IntVal b] = if a < b then Right TrueVal
93     ↳ else Right FalseVal
94 smallerThen [StringVal a, StringVal b] = if a < b then Right
95     ↳ TrueVal else Right FalseVal

```

```

87  smallerThen [_, _] = Right FalseVal
88  smallerThen _ = Left "Wrong number of arguments"
89
90  add :: Primitive
91  add [IntVal a, IntVal b] = Right (IntVal(a + b))
92  add [StringVal a, StringVal b] = Right (StringVal(a ++ b))
93  add [IntVal a, StringVal b] = Right (StringVal(show a ++ b))
94  add [StringVal a, IntVal b] = Right (StringVal(a ++ show b))
95  add [_, _] = Left "No Int or String"
96  add _ = Left "Wrong number of arguments"
97
98  mul :: Primitive
99  mul [IntVal a, IntVal b] = Right (IntVal(a*b))
100 mul [_, _] = Left "No Integer"
101 mul _ = Left "Wrong number of arguments"
102
103 sub :: Primitive
104 sub [IntVal a, IntVal b] = Right (IntVal(a-b))
105 sub [_, _] = Left "No Integer"
106 sub _ = Left "Wrong number of arguments"
107
108 modulo :: Primitive
109 modulo [IntVal a, IntVal b] = if b == 0 then Left "Division by
    ↪ Zero" else Right (IntVal(mod a b))
110 modulo [_, _] = Left "No Integer"
111 modulo _ = Left "Wrong number of arguments"
112
113 mkArray :: Primitive
114 mkArray [IntVal n] | n >= 0 = return $ ArrayVal (replicate n
    ↪ UndefinedVal)
115 mkArray _ = Left "Array() called with wrong number or type of
    ↪ arguments"
116
117 modifyEnv :: (Env -> Env) -> SubsM ()
118 modifyEnv f = SubsM $ \ (e, _) -> Right (((), f e)
119
120 putVar :: Ident -> Value -> SubsM ()
121 putVar name val = modifyEnv $ \e -> Map.insert name val e
122
123 getVar :: Ident -> SubsM Value
124 getVar name = SubsM $ \ (e, _) -> case Map.lookup name e of
125     Just v -> Right (v, e)
126     Nothing -> Left "No value
    ↪ found in map"

```

```

127
128 getFunction :: FunName -> SubsM Primitive
129 getFunction name = SubsM $ \ (e, p) -> case Map.lookup name p of
130                                     Just v -> Right (v, e)
131                                     Nothing -> Left "No value
                                                ↳ found in map"
132
133 evalExpr :: Expr -> SubsM Value
134 evalExpr Undefined = return UndefinedVal
135 evalExpr TrueConst = return TrueVal
136 evalExpr FalseConst = return FalseVal
137 evalExpr (Number a) = return $ IntVal a
138 evalExpr (String a) = return $ StringVal a
139 evalExpr (Var a) = getVar a
140 evalExpr (Array []) = return (ArrayVal [])
141 evalExpr (Array (a:ax)) = do
142     a <- evalExpr a
143     ArrayVal ax <- evalExpr (Array ax)
144     return (ArrayVal (a:ax))
145 evalExpr (Compr (ACBody e)) = evalExpr e
146 evalExpr (Compr (ACFor i e c)) = do
147     a <- evalExpr e
148     case a of
149         ArrayVal xa -> do
150             val <- mapM (\x -> do
151                 putVar i x
152                 evalExpr (Compr c)) xa
153             return (ArrayVal val)
154         StringVal xs -> do
155             (StringVal s) <- (\_ -> evalExpr (Compr c)) xs
156             return (StringVal s)
157         _ -> fail "FOR needs an array or string"
158
159 evalExpr (Compr (ACIf e c)) = do
160     a <- evalExpr e
161     case a of
162         TrueVal -> evalExpr (Compr c)
163         FalseVal -> return (ArrayVal [])
164         _ -> fail "IF needs a boolean"
165
166 evalExpr (Call a b) = do
167     f <- getFunction a
168     ArrayVal bv <- evalExpr (Array b)
169     case f bv of

```

```
170     Right r -> return r
171     Left l -> fail l
172
173 evalExpr (Assign a b) = do
174   v <- evalExpr b
175   putVar a v
176   return v
177
178 evalExpr (Comma a b) = do
179   _ <- evalExpr a
180   evalExpr b
181
182 runExpr :: Expr -> Either Error Value
183 runExpr expr = case runSubsM (evalExpr expr) initialContext of
184     Right r -> Right (fst r)
185     Left l -> Left l
```

---