AP Assignment 2:
Monadic Programming

Nikolaj Gade (qhp695), Sebastian Larsen Prehn (vpz655), Alba Dekens (dmb904)

1 Introduction

The implemented code compiles and passes all the tests which we wrote for it. Our testing is not exhaustive, but tests the general functionality of all the functions implemented. We sadly did not have the time to accurately showcase opposite behavior as instructed for the third question of the **Questions** section, so have opted to instead submit the code at its latest working iteration that satisfies all the other requirements as we have understood them.

2 Printing

We successfully added the feature of being able to print values in our APL program. We added the new type *State* to hold a list of strings, and then added the helper-function `evalPrint` to help with the logic of adding an entry to our new type *State*

```haskell
evalPrint :: String → EvalM ()
evalPrint a = EvalM $ \_env state → (state ++ [a], Right ())
```

We modified *EvalM* to handle this new *State*

```haskell
newtype EvalM a = EvalM (Env → State → (State, Either Error a))
```

and modified the Applicative to include the *State* and the Monad

```haskell
instance Monad EvalM where
  EvalM x >>= f = EvalM $ \env state →
    case x env state of
      (state', Left err) → (state', Left err)
      (state', Right x') →
        let EvalM y = f x'
         in y env state'
```

where we check to ensure that the input is valid, and otherwise return an error.

Finally, we made sure to extend `eval` in order to check the data type and return the textual representation depending on if we get a integer, boolean or function.

```haskell
eval (Print s e1) = do
  v1 <- eval e1
  case v1 of
    (ValInt i) → do
      evalPrint (s ++ ": " ++ (show i))
      pure $ v1
    (ValBool b) → do
      evalPrint (s ++ ": " ++ (show b))
      pure $ v1
    (ValFun _ _ _) → do
      evalPrint (s ++ ": #<fun>")
```

```
160       pure $ v1
```

3 Key-value store

We redefine the `State` type to be a tuple of a list of key-value bindings and the print list:

```
20  type State = ([(Val,Val)],[String])
```

We then define some helper functions, similar to the env functions, to help with manipulating the key-value store.

```
90  evalKvGet :: Val → EvalM Val
91  evalKvGet a = EvalM $ \_env (k,s) → do
92    case lookup a k of
93      (Just v) → ((k,s), Right v)
94      (Nothing) → ((k,s), Left ("Invalid key: " ++ (show a)))
95
96  evalKvPut :: Val → Val → EvalM ()
97  evalKvPut a b = EvalM $ \_env (k,s) → (((a,b):k,s), Right ())
```

Using those helper functions, we write our eval function for KvPut and KvGet:

```
161 eval (KvPut e1 e2) = do
162   v1 <- eval e1
163   v2 <- eval e2
164   evalKvPut v1 v2
165   pure $ v2
166 eval (KvGet e) = do
167   v <- eval e
168   evalKvGet v
```

4 Type checking

We started with the monad definition, which is mostly similar to the `EvalM`:

```
15  newtype CheckM a = CheckM (Vars → Either Error a)
16
17  instance Functor CheckM where
18    fmap = liftM
19
20  instance Applicative CheckM where
21    pure x = CheckM $ \_vars → Right x
22    (<*>) = ap
23
24  instance Monad CheckM where
25    CheckM x >>= f = CheckM $ \vars →
26      case x vars of
27        Left err → Left err
28        Right x' →
29          let CheckM y = f x'
30           in y vars
```

We then wrote some helper functions. One that returns whether an element is in a list, 3 that are similar to the env helper functions from eval, and 2 which check multiple expressions:

```
30  inList :: VName → [VName] → Bool
31  inList _ [] = False
32  inList x (y : ys) = if x == y then True else inList x ys
33
```

```haskell
askVars :: CheckM Vars
askVars = CheckM $ \vars → Right vars

varsExtend :: VName → Vars → Vars
varsExtend v vars = v : vars

localVars :: (Vars → Vars) → CheckM a → CheckM a
localVars f (CheckM m) = CheckM $ \vars → m (f vars)

checkBinOp :: Exp → Exp → CheckM ()
checkBinOp e1 e2 = do
  _ <- check e1
  check e2

checkTriOp :: Exp → Exp → Exp → CheckM ()
checkTriOp e1 e2 e3 = do
  _ <- check e1
  _ <- check e2
  check e3
```

Then, the `check` functions. The constants return a base `CheckM`, and all the expressions that don't deal with variables are checked recursively without any further logic. The 3 expressions that utilize variables have checks based on the helper function.

```haskell
check :: Exp → CheckM ()
check (CstInt _) = CheckM $ \_vars → pure ()
check (CstBool _) = CheckM $ \_vars → pure ()
check (Add e1 e2) = checkBinOp e1 e2
check (Sub e1 e2) = checkBinOp e1 e2
check (Mul e1 e2) = checkBinOp e1 e2
check (Div e1 e2) = checkBinOp e1 e2
check (Pow e1 e2) = checkBinOp e1 e2
check (Eql e1 e2) = checkBinOp e1 e2
check (If e1 e2 e3) = checkTriOp e1 e2 e3
check (Apply e1 e2) = checkBinOp e1 e2
check (TryCatch e1 e2) = checkBinOp e1 e2
check (Print _ e1) = check e1
check (KvGet e1) = check e1
check (KvPut e1 e2) = checkBinOp e1 e2
check (Var v) = do
  vars <- askVars
  if inList v vars
    then pure ()
    else CheckM $ \_vars → Left $ "Variable not in scope: "++v
check (Let var e1 e2) = do
  _ <- check e1
  localVars (varsExtend var) $ check e2
check (Lambda var e1) = do
  localVars (varsExtend var) $ check e1
```

Finally, we constructed the functions that run the type checking monad:

```haskell
runCheck :: CheckM a → Either Error a
runCheck (CheckM m) = m varsEmpty

checkExp :: Exp → Maybe Error
checkExp e =
  case runCheck $ check e of
    Left err → Just err
    _ → Nothing
```

The assignment handout does not specify how we should deal with try-catch, but we should ideally check both and only return error if both conditions fail. Our code does not do this, but it would be ideal.

5 Questions

1. Our code appends each newly printed string to the end of the list. This is done in linear time, based on the length of the list. However, if we constructed the list in reverse by appending to the beginning of the list, this could be done in constant time.

2. In our implementation, if a program keeps updating the same key with `KvPut`, then we append it to the top of the list, thus making it come before the older entry in the same list. Thus, in our implementation the memory usage does increase, as the list grows with each usage of `KvPut`. If we had instead implemented our program to lookup the key and replace the key value with the newer key value, then we would have had the same memory usage for each update of the same key, but it would be more computationally expensive. However, this would generally have been preferable to the increased memory usage.

3. If *e1* fails after replacing a value in a key-value pair, that effect is visible in *e2*, and we test for it in `trycatch(keyvalue)`. In order to change the behavior of the code, we would have to modify `catch` in order to get the opposite behavior. We did not find a solution that worked in time for this, however.