Assignment 2

Advance Programming

Name: Kazi Sohrab Uddin Titu, Group Number: 58

1 Introduction:

This assignment consists of some new features like state management, support for printing values, a key-value store and a type checker. I have tried some test cases and some with propagating errors also. I haven't got any major issue related to this solution. However, some areas can be improved like optimizations of how the key-value store is managed. I have tried to ensure coverage of different cases. Though there are some edge cases that could be refined further but I tried to cover them up. To run test cases, run cabal test in the terminal.

2 Task: Printing

Solution Functionality:
The interpreter was extended to support state management for handling printed strings. During the evaluation of expressions a log of printed values is maintained. EvalM monad was implemented to tracks both the environment and printed strings.

Failures or Limitations:
The printing functionality works as expectation and passes all tests. However, there is one concern that the printed strings are merged in a list but it can grow in size for large programs, though it was not a major concern for the current scope.

Potential Improvements:
As my opinion I think I could implement more efficient logging mechanisms, especially if a program involves heavy printing operations. Alternatively, optimizing how the log is handled could improve performance in more complex scenarios.

3 Task: Key-value store

Solution Functionality:
I have added key-value store to allow arbitrary key-value pairs to be stored and retrieved during evaluation. The KvPut and KvGet expressions were implemented to get this functionality. The store is tracked as part of the state managed by the EvalM monad.

Failures or Limitations:
I have given some test cases with propagating error but I think it could suffer performance issues when a large number of key-value pairs will be appended. As the store is represented as a list, the complexity of searching and append operations are linear.

Potential Improvements:
I think we could use more efficient data structure like Map to optimize the append and searching operations. Map would allow for constant memory usage when updating the same key.

4 Task: Type checking

Solution Functionality:
The aim of this type checker is to make sure that all variables are properly scoped and there is no undeclared variable. CheckM monad implemented to track variable scopes and report errors when variables are not in scope.

Failures or Limitations:
It has passed all positive and negative test cases I gave. It can detect scope violations and reports errors.

Potential Improvements:
I think we could improve optimization in how scopes are managed like in nested expressions. We can also improve to handle complex expressions.

5 Questions:

1. What is the asymptotic complexity of your implementation of Print? Can this be improved?

The complexity of the print operation is linear with respect to the number of printed strings. It depends on the size of the first list. We can use DList as an improvement to get a constant-time append operations.

2. In your implementation, if a program keeps updating the same key with KvPut, does the memory usage of the program remain constant or does it increase?

In my current implementation, each KvPut creates a new entry in the store even if the key already exists which increases the memory usage. Though old key-value pair is removed but overall size of the list grows.

3. In TryCatch e1 e2, if e1 fails after performing some effects, are those effects visible in e2, and which of your tests demonstrate this? If you wanted different behaviour (either making the effects visible or invisible), what would you need to change in your implementation? Show exactly what the new code would look like.

Yes, in the current implementation, the effects performed by e1 are visible in e2. As an example, if e1 prints a string before failing, that printed string remains in the log, and e2 can execute afterward. This behavior is demonstrated in below test case:

```
testCase "TryCatch" $
  eval' (TryCatch (Div (CstInt 7) (CstInt 0)) (CstBool True))
  @?= Right (ValBool True)
```

If I wanted different behavior like making the effects invisible, I have to ensure that e2 executes with the original state. Restoring the original state before evaluating e2.

```
catch :: EvalM a -> EvalM a -> EvalM a
catch (EvalM m1) (EvalM m2) = EvalM $ \env store ->
  let (log1, res1, store1) = m1 env store
  in case res1 of
      Left _ -> let (log2, res2, store2) = m2 env store
                in (log1 ++ log2, res2, store)
      Right val -> (log1, Right val, store1)
```