Final Exam: The Beginning of the End

CS 421 – Spring 2016
Revision 0.6

Assigned TODO
Due TODO

1 Change Log

1.0 Initial Release.

2 Objectives

Your objective is to write an interpreter for a contrived language in order to demonstrate knowledge of the internals of an interpreter.

3 Our Datatypes

There are three datatypes we'll be using: `Stmt`, `Exp`, and `Val`. In particular, `Stmt`s are statements which can modify the environment and potentially have side effects, `Exp`s are expressions which cannot have side effects, and `Val`s are our values.

3.1 Statements

Here are the different `Stmt`s that we can have:

- `SeqStmt s1 s2`, which represents sequential execution (i.e., $S_1;\ S_2$)
- `SetStmt x e`, which represents $x := e$
- `IfStmt e1 s1 s2`, which represents `if` e_1 `then` $\{\ S_1\ \}$ `else` $\{\ S_2\ \}$
- `WhileStmt e s`, which represents `while` e `do` $\{\ S\ \}$ `od`
- `ForStmt x e1 e2 s`, which represents `for` x `from` e_1 `to` e_2 `do` $\{\ S\ \}$ `od`
- `TryStmt s1 s2`, which represents `try` $\{\ S_1\ \}$ `on exception` $\{\ S_2\ \}$

```
data Stmt = SeqStmt Stmt Stmt
          | SetStmt String Exp
          | IfStmt Exp Stmt Stmt
          | WhileStmt Exp Stmt
          | ForStmt String Exp Exp Stmt
          | TryStmt Stmt Stmt
    deriving Eq
```

```
instance Show Stmt where
  show (SeqStmt s1 s2)         = show s1 ++ "; " ++ show s2
  show (SetStmt x e)           = x ++ " := " ++ show e
  show (IfStmt e1 s1 s2)       = "if (" ++ show e1 ++ ")"
                                   ++ " then { " ++ show s1 ++ " }"
                                   ++ " else { " ++ show s2 ++ " }"
  show (WhileStmt e1 s1)       = "while (" ++ show e1 ++ ")"
                                   ++ "do { " ++ show s1 ++ " } od"
  show (ForStmt x e1 e2 s1)    = "for " ++ x ++ " from " ++ show e1 ++ " to " ++ show e2
                                   ++ " do { " ++ show s1 ++ "} od"
  show (TryStmt s1 s2)         = "try { " ++ show s1 ++ " }"
                                   ++ "on exception { " ++ show s2 ++ " }"
```

3.2 Expressions

Here are the different Exps that we can have:

- IntExp n, which represents an integer n,
- BoolExp b, which represents a boolean b,
- ExnExp exn, which represents an exception with an error exn,
- VarExp x, which represents a variable x,
- IfExp e1 e2 e3, which represents an if expression,
- LetExp xs exp, which represents a let expression with simultaneous bindings (e.g., let [$x_1 := e_1$, ... , $x_n := e_n$] in exp),
- IntOpExp op e1 e2, which represents $e1$ *op* $e2$ (where *op* is an integer operation like $+$), and
- CompOpExp op e1 e2, which represents $e1$ *op* $e2$ (where *op* is a boolean comparison operator like $\leq$).

```
data Exp = IntExp Integer
         | BoolExp Bool
         | ExnExp String
         | VarExp String
         | IfExp Exp Exp Exp
         | LetExp [(String, Exp)] Exp
         | IntOpExp String Exp Exp
         | CompOpExp String Exp Exp
    deriving Eq

instance Show Exp where
  show (IntExp n)            = show n
  show (BoolExp b)           = show b
  show (ExnExp exn)          = "exception: " ++ exn
  show (VarExp s)            = s
  show (IfExp e1 e2 e3)      = "if " ++ show e1
                                 ++ " then " ++ show e2
                                 ++ " else " ++ show e3
  show (LetExp xs exp)       = let ls = Prelude.map (\(x, y) -> x ++ " := " ++ show y) xs
                               in  "let [" ++ intercalate ", " ls ++ "] in " ++ show exp
  show (IntOpExp op e1 e2)   = show e1 ++ " " ++ op ++ " " ++ show e2
  show (CompOpExp op e1 e2) = show e1 ++ " " ++ op ++ " " ++ show e2
```

3.3 Values

And finally, there are 3 different `Val`s that we can have:

- `IntVal n`, which represents an integer n,
- `BoolVal b`, which represents a boolean b, and
- `ExnVal exn`, which represents an exception with an error exn.

```
data Val = IntVal Int
         | BoolVal Bool
         | ExnVal String
    deriving Eq

instance Show Val where
    show (IntVal n)   = show n
    show (BoolVal b)  = show b
    show (ExnVal s)   = "exception: " ++ s
```

As you may surmise from reading the above, we have provided you with `Show` instances for each of the given datatypes.

3.4 Type Synonyms

We have also defined type synonyms to make things a little clearer:

```
type Env    = H.HashMap String Int
type Store  = H.HashMap Int Val
type Result = (Env, Store)
```

In particular:

- `Env` is our environment which maps a variable name (i.e., a string) to a location (i.e., an integer) in our `Store`,
- `Store` is our "memory" which maps a location in memory (i.e., an integer) to a `Val`,
- `Result`, which is a tuple containing an `Env` and a `Store`.

4 The Evaluator & The Executor

You will be working with two functions: `eval`, which is responsible for expressions like $(3 + 4)$, and `exec`, which is responsible for executing statements potentially with side effects.

In particular:

```
eval :: Exp -> Env -> Store -> Val
exec :: Stmt -> Env -> Store -> Either Val Result
```

Notice that `exec` can return an `Either Val Result`. In particular, if `exec` returns a `Left val`, it is interpreted as an error and the error is propagated up. If `exec` returns a `Right (env, mem)`, it is interpreted as a successful operation, and execution continues with the new environment and store.

This means that `exec` is strict with exceptions; once it sees an exception `exec` will propagate it up and halt execution. On the other hand, `eval` is lazy with exceptions; if `eval` sees an exception it will continue evaluating, only propagating the exception up if necessary.

5 The Code

You may be interested in the following files:

- `app/Interpreter.hs`: This is where the interpreter lives (i.e., your code goes here).
- `src/Lib.hs`: This is where our defined types live.
- `test/Tests.hs`: This is where our given tests live.
- `test/StudentTests.hs`: This is where your tests should live.

6 Logistics / Handing In

A subset of the problems below will be given to you on your final exam. You are allowed to ask course staff questions either during office hours or on Piazza about how to go about working on the problems, but they will not be answering questions on correctness/compilation errors/etc. Please also note that after the final exam period begins, course staff will no longer answer questions specifically about the exam. Conceptual questions, however, will be answered.

You will have access to `stack` during your exam. You will have to run a setup script (documentation coming soon!) prior to starting the question on your exam.

7 Testing

To test your code, you may run `stack test`. For test cases you have not passed, the following output will be displayed:

```
FAILED: x := (let [y := 7] in y)
              Expected: Right (fromList [("x",1)],fromList [(1,7)])
              Calculated: Right (fromList [("x",1)],fromList [(1,0)])
```

In particular, the test case failed is the first line, the expected output the second line, and the output from your implementation the third line.

Please note that the given test cases do not constitute a complete test suite. You are encouraged to write your own tests to verify that your code meets the given specifications.

8 Problems

Problem 1. Modify `eval` to handle `IntExp`, `BoolExp`, and `ExnExp`.

$$\frac{}{(n, env, mem) \Downarrow n}\; n \in \mathbb{Z} \text{ (set of integers)}$$

$$\frac{}{(b, env, mem) \Downarrow b}\; b \in \texttt{Bool}$$

$$\frac{}{(\texttt{ExnExp exn}, env, mem) \Downarrow \texttt{ExnVal exn}}$$

```
Main> eval (IntExp 0) env1 mem1
0
Main> eval (BoolExp True) env1 mem1
True
Main> eval (ExnExp "good luck") env1 mem1
exception: good luck
```

Problem 2. In order to do anything more complex, we need to be able to look things up in our environment/store. Modify the function `insertEnv` such that if a variable `x` is given, if it is already in the environment, will update the value in the store. Otherwise, we create a mapping from `x` to a new address `freshAddr` in the environment, and a mapping from `freshAddr` to the value `v` in the store.

In particular:

```
insertEnv :: String -> Val -> Env -> Store -> Result
insertEnv x v env mem = fixMe
    where freshAddr = (H.size mem) + 1

*Main> env1
fromList [("x",1)]
*Main> mem1
fromList [(1,0)]
*Main> insertEnv "x" (IntVal 5) env1 mem1
(fromList [("x",1)],fromList [(1,5)])
*Main> insertEnv "y" (IntVal 5) env1 mem1
(fromList [("x",1),("y",2)],fromList [(1,0),(2,5)])
```

Problem 3. Modify `eval` to handle `VarExp`.

$$\frac{}{(x, env, mem) \Downarrow v}\; x \in env;\; env(x) = n;\; n \in mem;\; mem(n) = v$$

$$\frac{}{(x, env, mem) \Downarrow \texttt{ExnVal "No match in environment."}}\; x \notin env$$

$$\frac{}{(x, env, mem) \Downarrow \texttt{ExnVal "Store error."}}\; x \in env;\; env(x) = n;\; n \notin mem$$

(Note that the last case technically can't happen because our stores are well-formed; it is here purely for completeness.)

```
Main> eval (VarExp "x") env1 mem1
0
```

Problem 4. Modify `eval` to handle `IfExp`.

$$\frac{(e_1, env, mem) \Downarrow \texttt{true} \quad (e_2, env, mem) \Downarrow v_f}{(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, env, mem) \Downarrow v_f}$$

$$\frac{(e_1, env, mem) \Downarrow \texttt{false} \quad (e_3, env, mem) \Downarrow v_f}{(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, env, mem) \Downarrow v_f}$$

$$\frac{(e_1, env, mem) \Downarrow \texttt{ExnVal exn}}{(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, env, mem) \Downarrow \texttt{ExnVal exn}}$$

$$\frac{(e_1, env, mem) \Downarrow v}{(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, env, mem) \Downarrow \texttt{ExnVal "Not a boolean guard."}}\; v \notin \texttt{Bool}$$

```
Main> eval
  (IfExp (CompOpExp "==" (IntExp 44) (IntExp 0)) (IntExp 44) (IntExp 0)) env1 mem1
0
```

Problem 5. Modify `eval` to handle `LetExp`. (Hint: A function you wrote earlier might be of use here.)

$$\frac{(e_1, env, mem) \Downarrow v_1 \quad ... \quad (e_n, env, mem) \Downarrow v_n \quad (e, env', mem') \Downarrow v_f}{(\texttt{let [}x_1 := e_1, ..., x_n := e_n\texttt{] in e}, env, mem) \Downarrow v_f}$$

where $m_1, \ldots, m_n$ are memory locations such that:

$$env' = \{x_1 \mapsto m_1, ..., x_n \mapsto m_n\} + env$$
$$mem' = \{m_1 \mapsto v_1, ..., m_n \mapsto v_n\} + mem$$

```
Main> eval (LetExp [("x", (IntExp 7))] (IntExp 5)) env1 mem1
5
```

Problem 6. Modify `eval` to handle `IntOpExp` and `CompOpExp`. The below rules apply for both expressions.

$$\frac{(e_1, env, mem) \Downarrow v_1 \quad (e_2, env, mem) \Downarrow v_2}{(e_1 \oplus e_2, env, mem) \Downarrow v_1 \oplus v_2}$$

$$\frac{(e_1, env, mem) \Downarrow v_1}{(e_1 \oplus e_2, env, mem) \Downarrow \texttt{ExnVal "Invalid integer operation."}} \; v_1 \notin \mathbb{Z} \text{ (set of integers)}$$

$$\frac{(e_1, env, mem) \Downarrow v_1 \quad (e_2, env, mem) \Downarrow v_2}{(e_1 \oplus e_2, env, mem) \Downarrow \texttt{ExnVal "Invalid integer operation."}} \; v_2 \notin \mathbb{Z} \text{ (set of integers)}$$

$$\frac{(e_1, env, mem) \Downarrow v_1 \quad (e_2, env, mem) \Downarrow v_2}{(e_1 \oplus e_2, env, mem) \Downarrow \texttt{ExnVal "Invalid integer operation."}} \; \oplus \text{ not a valid operator}$$

```
Main> eval (IntOpExp "+" (IntExp 40) (IntExp 4)) env1 mem1
44
Main> eval (CompOpExp "==" (IntExp 44) (IntExp 0)) env1 mem1
False
```

Problem 7. Modify `exec` to handle `SeqStmt`.

$$\frac{(S_1, env, mem) \Downarrow (env', mem') \quad (S_2, env', mem') \Downarrow (env'', mem'')}{(S_1;\ S_2, env, mem) \Downarrow (env'', mem'')}$$

$$\frac{(S_1, env, mem) \Downarrow \texttt{ExnVal exn}}{(S_1;\ S_2, env, mem) \Downarrow \texttt{ExnVal exn}}$$

$$\frac{(S_1, env, mem) \Downarrow (env', mem') \quad (S_2, env', mem') \Downarrow \texttt{ExnVal exn}}{(S_1;\ S_2, env, mem) \Downarrow \texttt{ExnVal exn}}$$

```
Main> exec (SeqStmt (SetStmt "f" (IntExp 5)) (SetStmt "f" (IntExp 7))) env1 mem1
Right (fromList [("x",1),("f",2)], fromList [(1,IntVal 0),(2,IntVal 7)])
```

Problem 8. Modify `exec` to handle `SetStmt`.

$$\frac{(e, env, mem) \Downarrow v}{(x := e, env, mem) \Downarrow (env', mem')} \text{ where } (env', mem') = \texttt{insertEnv}\ x\ v\ env\ mem$$

$$\frac{(e, env, mem) \Downarrow \texttt{ExnVal exn}}{(x := e, env, mem) \Downarrow \texttt{ExnVal exn}}$$

```
Main> exec (SetStmt "x" (LetExp [("y", IntExp 7)] (VarExp "y"))) env1 mem1
Right (fromList [("x",1)],fromList [(1,IntVal 7)])
```

Problem 9. Modify `exec` to handle `IfStmt`.

$$\frac{(e, env, mem) \Downarrow \texttt{true} \quad (S_1, env, mem) \Downarrow (env', mem')}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, env, mem) \Downarrow (env', mem')}$$

$$\frac{(e, env, mem) \Downarrow \texttt{true} \quad (S_1, env, mem) \Downarrow \texttt{ExnVal exn}}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, env, mem) \Downarrow \texttt{ExnVal exn}}$$

$$\frac{(e, env, mem) \Downarrow \texttt{false} \quad (S_2, env, mem) \Downarrow (env', mem')}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, env, mem) \Downarrow (env', mem')}$$

$$\frac{(e, env, mem) \Downarrow \texttt{false} \quad (S_2, env, mem) \Downarrow \texttt{ExnVal exn}}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, env, mem) \Downarrow \texttt{ExnVal exn}}$$

$$\frac{(e, env, mem) \Downarrow \texttt{ExnVal exn}}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, env, mem) \Downarrow \texttt{ExnVal exn}}$$

$$\frac{(e, env, mem) \Downarrow v}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, env, mem) \Downarrow \texttt{ExnVal "Not a boolean guard."}}\ v \notin \texttt{Bool}$$

```
Main> exec
  (IfStmt (CompOpExp "==" (VarExp "x") (IntExp 0))
    (SetStmt "r" (BoolExp True)) (SetStmt "r" (BoolExp False)))
  env1 mem1
Right (fromList [("x",1), ("r",2)],fromList [(1,IntVal 0), (2, BoolVal True)])
```

Problem 10. Modify `exec` to handle `WhileStmt`.

$$\frac{(e, env, mem) \Downarrow \texttt{true} \quad (S, env, mem) \Downarrow (env', mem') \quad (\texttt{while}\ e\ \texttt{do}\ S\ \texttt{od}, env', mem') \Downarrow (env'', mem'')}{(\texttt{while}\ e\ \texttt{do}\ S\ \texttt{od}, env, mem) \Downarrow (env'', mem'')}$$

$$\frac{(e, env, mem) \Downarrow \texttt{false}}{(\texttt{while}\ e\ \texttt{do}\ S\ \texttt{od}, env, mem) \Downarrow (env, mem)}$$

$$\frac{(e, env, mem) \Downarrow \texttt{ExnVal exn}}{(\texttt{while}\ e\ \texttt{do}\ S\ \texttt{od}, env, mem) \Downarrow \texttt{ExnVal exn}}$$

$$\frac{(e, env, mem) \Downarrow v}{(\texttt{while}\ e\ \texttt{do}\ S\ \texttt{od}, env, mem) \Downarrow \texttt{ExnVal "Not a boolean guard."}}\ v \notin \texttt{Bool}$$

```
Main> exec
  (WhileStmt (CompOpExp "<=" (VarExp "x") (IntExp 5))
    (SetStmt "x" (IntOpExp "+" (VarExp "x") (IntExp 1))))
  env1 mem1
Right (fromList [("x",1)],fromList [(1,IntVal 6)])
```

Problem 11. Modify `exec` to handle `ForStmt`.

$$\frac{(\texttt{x := } e_1\texttt{; while } (\texttt{x} < e_2) \texttt{ do } S \texttt{ od}, env, mem) \Downarrow (env', mem')}{(\texttt{for } x \texttt{ from } e_1 \texttt{ to } e_2 \texttt{ do } S \texttt{ od}, env, mem) \Downarrow (env', mem')}$$

$$\frac{(\texttt{x := } e_1\texttt{; while } (\texttt{x} < e_2) \texttt{ do } S \texttt{ od}, env, mem) \Downarrow \texttt{ExnVal exn}}{(\texttt{for } x \texttt{ from } e_1 \texttt{ to } e_2 \texttt{ do } S \texttt{ od}, env, mem) \Downarrow \texttt{ExnVal exn}}$$

```
Main> exec
  (ForStmt "x" (IntExp 0) (IntExp 5)
    (SetStmt "x" (IntOpExp "+" (VarExp "x") (IntExp 1))))
  env1 mem1
Right (fromList [("x",1)],fromList [(1,IntVal 5)])
```

Problem 12. Modify `exec` to handle `TryStmt`.

$$\frac{(S_1, env, mem) \Downarrow (env', mem')}{(\texttt{try } S_1 \texttt{ on exception } S_2, env, mem) \Downarrow (env', mem')}$$

$$\frac{(S_1, env, mem) \Downarrow \texttt{ExnVal exn} \quad (S_2, env, mem) \Downarrow (env', mem')}{(\texttt{try } S_1 \texttt{ on exception } S_2, env, mem) \Downarrow (env', mem')}$$

$$\frac{(S_1, env, mem) \Downarrow \texttt{ExnVal exn1} \quad (S_2, env, mem) \Downarrow \texttt{ExnVal exn2}}{(\texttt{try } S_1 \texttt{ on exception } S_2, env, mem) \Downarrow \texttt{ExnVal exn2)}}$$

```
Main> exec (TryStmt (SetStmt "x" (IntExp 5)) (SetStmt "y" (IntExp 7))) env1 mem1
Right (fromList [("x",1)],fromList [(1,IntVal 5)])
```