MP 2: Writing an Interpreter

CS 421

Revision 1.0

Assigned February 10, 2016
Due February 24, 2016

1 Objectives

The objective for this MP is to write an interpreter for a language with both expressions and statements. In particular, your job is to implement the "E" (evaluate) part of the read-eval-print loop (`REPL`).

2 Getting Set Up

You might need to install the following libraries:

- `parsec` for `Text.ParserCombinators.Parsec`
- `unordered-containers` for `Data.HashMap.Strict`

To install them, run `cabal install parsec unordered-containers`.

3 Given Code

We have provided you with some stubbed code to help get you started. In particular, we have provided the algebraic data types that you'll need to utilize in your evaluator, along with the portions to handle reading and printing (and of course, looping). Your task for this MP is to complete the `REPL` by finishing the interpreter.

3.1 Data Types

We have three different data types that we need to make our interpreter: one for statements, one for expressions, and one for values.

3.1.1 Statements

A statement is an operation intended to yield a side effect. We have `SetStmt` for variable assignment, `PrintStmt` for printing, `QuitStmt` to exit the interpreter, `IfStmt` for conditional statements, `ProcedureStmt` for procedure definitions, `CallStmt` for procedure calls, and `SeqStmt` to sequence statements (just like the semicolon in some languages). We have taken care of `QuitStmt` in the `REPL` and `PrintStmt` in `eval`; the rest is part of your task for this MP.

```
data Stmt = SetStmt String Exp
          | PrintStmt Exp
          | QuitStmt
          | IfStmt Exp Stmt Stmt
          | ProcedureStmt String [String] Stmt
          | CallStmt String [Exp]
          | SeqStmt [Stmt]
   deriving Show
```

3.1.2 Expressions

Expressions are statements that are evaluated to become values. We have `IntExp` for integers, `BoolExp` for booleans, `FunExp` for functions, `LetExp` for let expressions, `AppExp` for function applications, `IfExp` for if expressions, `IntOpExp` for integer operations (such as addition), `BoolOpExp` for boolean operations (such as *and* and *or*), `CompOpExp` for comparisons between integers, and `VarExp` for variables.

```
data Exp = IntExp Int
         | BoolExp Bool
         | FunExp [String] Exp
         | LetExp [(String,Exp)] Exp
         | AppExp Exp [Exp]
         | IfExp Exp Exp Exp
         | IntOpExp String Exp Exp
         | BoolOpExp String Exp Exp
         | CompOpExp String Exp Exp
         | VarExp String
   deriving Show
```

3.1.3 Values

We have a few kinds of values as well: `IntVal` and `BoolVal` for integers and booleans, and `CloVal` to represent closures. We also have a value for exceptions which we'll call `ExnVal`.

Closures, as you may recall, have two parts: the function, and the environment from when we created the closure. They allow us to maintain the state of the program from when it was created. For example, if there were global variables that existed at the time, we want to have access to the original copies of them in case they're referenced in the function and are possibly modified after the creation of the closure. Here, we split up the function part of the closure into two parts: the parameters, and the function body.

```
data Val = IntVal Int
         | BoolVal Bool
         | CloVal [String] Exp Env
         | ExnVal String

instance Show Val where
   show (IntVal i) = show i
   show (BoolVal i) = show i
   show (CloVal f penv env) = "<" ++ show f ++ ", "
                               ++ show penv ++ ", "
                               ++ show env ++ ">"
   show (ExnVal s) = "\nexn: " ++ s
```

3.2 Environments

Environments are a series of mappings from variables to values; these values might be an actual value, like 3 or `True`, or it might be something as "abstract" as a closure. In our world, we will be using hash maps to represent environments.

In this case, `Env` is the value environment; it tells us the values contained in whatever variables we are using. `PEnv` is the procedure environment; it gives us the procedure body for any procedure we want to call. The `Result` type contains the result of executing a statement - in particular, a triple containing the output that we wish to display from evaluating the statement, the procedure environment at that point, and the value environment at that point.

```
type Env = H.HashMap String Val
type PEnv = H.HashMap String Stmt

type Result = (String, PEnv, Env)
```

3.3 Operations

The following hash maps contain the operations we will want to use. We have a hash map for each set of operations: integer, boolean, and comparison.

```
intOps = H.fromList [ ("+", (+))
                    , ("-", (-))
                    , ("*", (*))
                    , ("/", div) ]
boolOps = H.fromList [ ("and", (&&))
                     , ("or", (||)) ]
compOps = H.fromList [ ("<", (<))
                     , (">", (>))
                     , ("<=", (<=))
                     , (">=", (>=))
                     , ("/=", (/=))
                     , ("==", (==)) ]
```

The "lift" functions unbox a `Val`, perform the operation we want, and then rebox the result in another `Val`. These functions make it easy for us to apply Haskell operations to our functions. You'll have to implement `liftCompOp` as part of this MP.

```
liftIntOp op (IntVal x) (IntVal y) = IntVal $ op x y
liftIntOp _ _ _ = ExnVal "Cannot lift"

liftBoolOp op (BoolVal x) (BoolVal y) = BoolVal $ op x y
liftBool _ _ _ = ExnVal "Cannot lift"

liftCompOp :: (Int -> Int -> Bool) -> Val -> Val -> Val
liftCompOp = fixMe
liftCompOp _ _ _ = ExnVal "Cannot lift"
```

3.4 The Evaluator

Here is the type declaration for the evaluator. The job of the evaluator is to evaluate expressions, like $(3+4)$ and `(true and true == false)`. You will be defining the entire function; we've provided you with the type declaration to get you started.

```
eval :: Exp -> Env -> Val
```

3.5 The Executor

Here is the code for the statement executor. Its job is to execute statements and perform whatever side-effecting operation was specified. Notice it takes both environments as arguments, not just the value environment. This is because its actions can modify not only the value environment, but also the procedure environment. Here, we define the `PrintStmt` case and save the rest for you.

```
exec :: Stmt -> PEnv -> Env -> Result
exec (PrintStmt e) penv env = (val, penv, env)
   where val = show $ eval e env
```

3.6 The REPL

Finally, there is `repl`. It calls the parser code (not shown here) to get the raw statement, parse it, and convert it into a `Stmt`. It then proceeds to call `exec` on the `Stmt`. You will not need to do anything with this, but you will want to understand how it works as you will use variations of it throughout the semester.

4 The Code

You will find three files in your repository:

- `Interpreter.hs`: This is where you'll be working. It contains `repl`, `exec`, and `eval`.
- `Parser.hs`: This contains the parser and the algebraic data types as given earlier.
- `Tests.hs`: This file contains some basic tests for you to run. You are expected to write more tests to ensure that your interpreter is complete and functions properly. We will not be grading these tests, but we will be running your code against a larger suite of tests.

  We've also provided you with some basic functions for testing:

  - `showStmt`: Takes a statement and (if parseable), returns the corresponding abstract syntax tree (i.e. the corresponding `Stmt`).

    ```
    *Tests> showStmt "print 1;"
    "PrintStmt (IntExp 1)"
    ```

  - `evalTester`: Returns the triple containing the output displayed in the interpreter, the procedure environment, and the value environment.

    ```
    *Tests> evalTester "print 1;"
    ("1",fromList [],fromList [])
    ```

  - `showAll`: Shows all the parsed statements in the list of tests we've given you.
  - `evaluateAll`: Evaluates all the parsed statements in the list of tests we've given you.
  - `testAll`: Compares all the output values with our expected output values.

5 Testing

5.1 Using The Interpreter

To use the interpreter, run `ghci Interpreter.hs`, then type in `main`.

Here is a sample compile and run:

```
$ ghci Interpreter.hs
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Parser           ( Parser.hs, interpreted )
[2 of 2] Compiling Interpreter      ( Interpreter.hs, interpreted )
Ok, modules loaded: Parser, Interpreter.
*Main> main
  ... other messages ...
Welcome to your interpreter!
> print 44;
44
```

Note that `$` is the UNIX prompt, and > is the interpreter prompt.

5.2 Automated Tests

We have written up and given you a (very, very basic) set of tests for you to test your code against. You are expected to write a set of tests yourself to ensure that your interpreter is correct. We'll be running your code against a larger set of tests.

```
$ ghci Tests.hs
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
[1 of 3] Compiling Parser           ( Parser.hs, interpreted )
[2 of 3] Compiling Interpreter      ( Interpreter.hs, interpreted )
[3 of 3] Compiling Tests            ( Tests.hs, interpreted )
Ok, modules loaded: Parser, Tests, Interpreter.
*Tests> showAll
  ... abstract syntax trees of statements ...
*Tests> evaluateAll
  ... triples of final states after evaluation ...
```

6 Handing In

You should commit a copy of your solution to your student repository in the `mp2-interpreter` folder.

To commit and push your work:

```
git add mp2.hs; git commit -m "handing in mp2"; git push
```

7 Your Task

Problem 1. Define `liftCompOp`.

```
:t liftCompOp
liftCompOp :: (Int -> Int -> Bool) -> Val -> Val -> Val
*Interpreter> liftCompOp (\x -> \y -> x == y) (IntVal 3) (IntVal 3)
True
```

Problem 2. In order to do any execution, we need to be able to handle sequential statements. In our world, this means being able to evaluate one or more statements. Define `SeqStmts` for `exec`.

$$\frac{(S_1, penv, env) \Downarrow (p_1, penv', env') \qquad (S_2, penv', env') \Downarrow (p_2, penv'', env'')}{(S_1; S_2, penv, env) \Downarrow (p_1 \;\; \texttt{++} \;\; p_2, penv'', env'')}$$

Problem 3. Before we can do any evaluation, we'll also need to define some basic expressions in `Exp` for `eval`. In particular, modify `eval` to handle both `IntExps` and `BoolExps`. Note that you should not write a function `constToVal`, but instead do the conversion in `eval` itself.

$$\frac{}{(t, env) \Downarrow \texttt{constToVal t}} \; t \text{ an integer/boolean constant}$$

```
*Interpreter> main
Welcome to your interpreter!
> print 1;
1
```

Problem 4. Now that we have some basic infrastructure set up, we'd like to work with more complex/interesting statements. To start, modify `exec` to handle `IfStmts`.

$$\frac{(e_1, env) \Downarrow \texttt{true} \qquad (S_2, penv, env) \Downarrow (p_2, penv', env')}{(\texttt{if } e_1 \texttt{ then } S_2 \texttt{ else } S_3, penv, env) \Downarrow (p_2, penv', env')}$$

$$\frac{(e_1, env) \Downarrow \texttt{false} \qquad (S_3, penv, env) \Downarrow (p_3, penv', env')}{(\texttt{if } e_1 \texttt{ then } S_2 \texttt{ else } S_3, penv, env) \Downarrow (p_3, penv', env')}$$

```
*Interpreter> main
Welcome to your interpreter!
> if true then print 1; else print 0; fi
1
```

Problem 5. Now, we want to be able to use variables. Modify `eval` to handle `VarExps`.

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

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

Problem 6. Now that we have variables, we want to be able to assign them values. Modify `exec` to handle `SetStmts`.

$$\frac{(e, env) \Downarrow v_1}{(\texttt{x := } e, penv, env) \Downarrow (\texttt{""}, penv, env + \{\texttt{x} \mapsto v_1\})}$$

```
*Interpreter> main
Welcome to your interpreter!
> if true then do x := 1; print x; od; else do x := 0; print x; od; fi;
1
```

Problem 7. Modify `eval` to handle `IntOpExp` so that we can evaluate arithmetic expressions.

$$\frac{(e_1, env) \Downarrow v_1 \qquad (e_2, env) \Downarrow v_2}{(e_1 \oplus e_2, env) \Downarrow (v_1 \oplus v_2)} \oplus \text{ an integer operation}$$

$$\frac{}{(e_1 \oplus e_2, env) \Downarrow \texttt{ExnVal "No matching operator"}} \oplus \text{ not a valid integer operation}$$

```
*Interpreter> main
Welcome to your interpreter!
> print 1+2;
3
```

Problem 8. Modify `eval` to handle `BoolOpExp` so that we can evaluate boolean expressions.

$$\frac{(e_1, env) \Downarrow v_1 \qquad (e_2, env) \Downarrow v_2}{(e_1 \oplus e_2, env) \Downarrow (v_1 \oplus v_2)} \oplus \text{ a boolean operation}$$

$$\frac{}{(e_1 \oplus e_2, env) \Downarrow \texttt{ExnVal "No matching operator"}} \oplus \text{ not a valid boolean operation}$$

```
*Interpreter> main
Welcome to your interpreter!
> if true and true then print 1; else print 0; fi;
1
```

Problem 9. Modify `eval` to handle `CompOpExp` so that we can evaluate integer comparison expressions.

$$\frac{(e_1, env) \Downarrow v_1 \qquad (e_2, env) \Downarrow v_2}{(e_1 \oplus e_2, env) \Downarrow (v_1 \oplus v_2)} \oplus \text{ an integer comparison operation}$$

$$\frac{}{(e_1 \oplus e_2, env) \Downarrow \texttt{ExnVal "No matching operator"}} \oplus \text{ not a valid integer comparison operation}$$

```
*Interpreter> main
Welcome to your interpreter!
> if (4>3) then print 1; else print 0; fi
1
```

Problem 10. Modify `eval` to handle `LetExp` so that we can properly handle scoped declarations.

$$\frac{(e_1, env) \Downarrow v_1 \quad ... \quad (e_n, env) \Downarrow v_n \quad (e_f, env' = env + \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}) \Downarrow v_f}{(\texttt{let [}x_1 := e_1, ..., x_n := e_n\texttt{]}\, e_f, env) \Downarrow v_f}\ n \geq 0$$

```
*Interpreter> main
Welcome to your interpreter!
> do x := let[x:=5] (if x > 3 then 1 else 0 fi) end; print x; od;
1
```

Problem 11. Modify `eval` to handle `FunExp`, allowing us to create functions. Note that this creates a closure, which encapsulates both the function definition and the environment at the time of creation.

$$\frac{}{(\texttt{fn}[x_1, ..., x_n]\ body\ \ \texttt{end;}, env) \Downarrow \langle [x_1, ..., x_n], body, env\rangle}$$

Problem 12. Modify `eval` to handle `AppExp`. This, in conjunction with `FunExp`, will allow us to do function application.

$$\frac{(e_1, env) \Downarrow v_1 \quad ... \quad (e_n, env) \Downarrow v_n \quad (body, env' + \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}) \Downarrow v_f}{(\texttt{apply}\ \ f(e_1, ..., e_n\texttt{);}, env) \Downarrow v_f}\ n \geq 0, f \in env$$

where $\quad env(f) = \quad \langle x_1, ..., x_n, body, env'\rangle$

$$\frac{}{(\texttt{apply}\ \ f(e_1, ..., e_n\texttt{);}, env) \Downarrow \texttt{ExnVal "No match in env"}}\ f \notin env$$

```
*Interpreter> main
Welcome to your interpreter!
> do f := fn[x,y] (x+y) end; print (apply f(1,2)); od;
3
```

Problem 13. Modify `exec` to handle `ProcedureStmt`. Procedures allows us to repeat code (much like a function would), but without the restoration of the old environment.

$$\frac{}{(\texttt{procedure}\ \ f\texttt{(}ps\texttt{)}\ \ body\texttt{;}\ \ \texttt{endproc;}, penv, env) \Downarrow (\texttt{""}, penv + \{f \mapsto (\texttt{ProcedureStmt}\ \ f\ \ ps\ \ body)\}, env)}$$

Problem 14. Modify `exec` to handle `CallStmt`. This allows us to actually use the procedures we've created.

$$\frac{A \quad (\mathit{body}, \mathit{penv}, \mathit{env}' = \mathit{env} + \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}) \Downarrow (p_f, \mathit{penv}', \mathit{env}'')}{(\texttt{call}\ f\,\texttt{(}e_1, ..., e_n\texttt{)}, \mathit{penv}, \mathit{env}) \Downarrow (p_f, \mathit{penv}', \mathit{env}'')}\ n \geq 0, f \in \mathit{penv}$$

$$\begin{aligned} \text{where} \qquad A &= \quad (p_1, \mathit{env}) \Downarrow v_1 \quad ... \quad (p_n, \mathit{env}) \Downarrow v_n \\ \mathit{penv}(f) &= \quad (\texttt{ProcedureStmt}\ f\ \mathit{ps}\ \mathit{body}) \end{aligned}$$

$$\frac{}{(\texttt{call}\ f\,\texttt{(}e_1, ..., e_n\texttt{)}, \mathit{penv}, \mathit{env}) \Downarrow \texttt{ExnVal "Procedure}\ f\ \texttt{undefined"}}\ f \notin \mathit{penv}$$

```
*Interpreter> main
Welcome to your interpreter!
> do procedure foo() print 1; endproc call foo(); od;
1
```

Problem 15. Modify `eval` to handle `IfExp`. This allows us to do things like call different functions given the result of evaluating a boolean guard.

$$\frac{(e_1, \mathit{env}) \Downarrow \texttt{true} \quad (e_2, \mathit{env}) \Downarrow v_2}{(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \mathit{env}) \Downarrow v_2}$$

$$\frac{(e_1, \mathit{env}) \Downarrow \texttt{false} \quad (e_3, \mathit{env}) \Downarrow v_3}{(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \mathit{env}) \Downarrow v_3}$$

```
*Interpreter> main
Welcome to your interpreter!
> do x := (if true then 1 else 0 fi); print x; od;
1
```