MP2 - Interpreter

- revision: 2.0

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.)

Goals

- Become familiar with environments and closures
- More practice with ADTs, pattern matching, recursion, and other functional programming concepts
- Learn how to use Hash Maps in Haskell

Useful Reading

We are using Data.HashMap.Strict for this MP and moving forward. It is a good idea to familiarize yourself with its interface. Of particular interest for this assignment are
`empty    :: HashMap k v`,
`fromList :: (Eq k, Hashable k) => [(k, v)] -> HashMap k v`,
`insert   :: (Eq k, Hashable k) => k -> v -> HashMap k v -> HashMap k v`, and
`lookup   :: (Eq k, Hashable k) => k -> HashMap k v -> Maybe v`.

Getting Started

Relevant Files

In the directory `src` you'll find `Lib.hs` with all the relevant code. In this file you will find all of the data definitions, the primitive function maps, the parser, stubbed out lifting functions, and stubbed out evaluation functions. In the directory `app` you will find `Main.hs` which contains the code for a REPL of our language. You are only responsible for making changes to the lifting and evaluation functions in `Lib.hs`.

Running Code

To run your code, start GHCi with `stack ghci`. From here, you can test individual functions, or you can run the REPL by calling `main`:

```
$ stack ghci
 ... More Output ...
Ok, modules loaded: Main.
*Main> main
Welcome to your interpreter!
> quit;
Bye!
("",fromList [],fromList [])
```

To run the REPL directly, build the executable with `stack build` and run it with `stack exec main`:

```
$ stack build
mp2-interpreter-0.1.0.0: build
Preprocessing executable 'main' for mp2-interpreter-0.1.0.0...
 ... More Output ...
$ stack exec main
Welcome to your interpreter!
> quit;
Bye!
```

You can exit the REPL with the command `quit;`.

Testing Your Code

As in MP1, you will be able to run the test-suite with `stack test`:

```
$ stack test
```

HOWEVER since this MP has a substantially more complicated test spec, (look upon it with horror if you dare) We have included two ways of running the tests. The first is

```
$ stack test interpreter\:test\:friendly-test
```

This will run the tests (first unit tests, then property based tests on randomly generated input) and output failures in a semi-readable manner. You should use this test-suite while solving the MP.

The second way is

```
$ stack test interpreter\:test\:grader-test
```

This is the test ultimately run by the grader on PrairieLearn. It will fail for a given exercise if any of the associated unit tests or property tests fail. This will not give very readable output in case of failure. Use this test-suite only once you have finished the MP and you are ready to submit it. We have included these tests so you can see exactly what gets run by the grader.

The friendly test spec can be found in `test/friendly/FriendlySpec.hs` The Unit tests are enumerated in `test/UnitTests.hs`. The property tests will most likely *not* be helpful, but they are located in `test/PropertyTests.hs`.

Given Code

You do not need to (and should not have to) change any of the following portions of the given code; however, it will be helpful to understand why each part of the code has been provided.

Data Types

The given code defines several data types for use by our interpreter. We have a datatype that represents values calculated during evaluation, a datatype to represent program expressions that can be evaluated, and a datatype to represent program statements that can be executed. We also define types (actually type synonyms) to represent environments, as well the results of statement executions.

Environments and Results

Environments are a series of mappings from identifiers (variable names, function names, etc) to "values" - "values" here possibly being actual values resulting from evaluating expressions, but which can be any Haskell datatype. For instance we may have a mapping of procedure names to procedure bodies.

Here we have declared `Env`, the type of a value environment, which maps the variables in scope to their current values. We also have `PEnv`, the type of a procedure environment, which maps procedure names to procedure bodies, for use when we want to call a procedure.

The `Result` type contains the result of executing a statement - 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.

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

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

Values

We have a few kinds of values: `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
    deriving (Eq)

instance Show Val where
    show (IntVal i) = show i
    show (BoolVal i) = show i
    show (CloVal xs body env) = "<" ++ show xs   ++ ", "
                                    ++ show body ++ ", "
                                    ++ show env  ++ ">"
    show (ExnVal s) = "exn: " ++ s
```

We've also defined a `Show` instance for `Val`, so that they can be pretty-printed by GHC and GHCi.

Expressions

Expressions 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 binary integer operations (such as addition), `BoolOpExp` for binary boolean operations (such as `&&` and `||`), `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, Eq)
```

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, executing one after the other (just like the semicolon in some languages).

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

Primitive Functions

The language has a number of primitive functions, such as addition and various comparison operators. The following map the names of those functions to Haskell functions which we can use to do the actual computations.

```
intOps :: H.HashMap String (Int -> Int -> Int)
intOps = H.fromList [ ("+", (+))
                    , ("-", (-))
                    , ("*", (*))
                    , ("/", (div))
                    ]

boolOps :: H.HashMap String (Bool -> Bool -> Bool)
boolOps = H.fromList [ ("and", (&&))
                     , ("or", (||))
                     ]

compOps :: H.HashMap String (Int -> Int -> Bool)
compOps = H.fromList [ ("<", (<))
                     , (">", (>))
                     , ("<=", (<=))
                     , (">=", (>=))
```

```
        , ("/=", (/=))
        , ("==", (==))
        ]
```

Parser

We have given you the entire parser this time around. The parser takes the command you type into the REPL (a `String`) and converts this string into a `Stmt` so that you can much more easily execute it. While it isn't important that you understand how the parser works right now, it may be interesting to take a look at it to see how much you can figure out. At the very least, this parser will be a good example for you to look at for future assignments, so keep that in mind.

It is worth noting that this language does not have the same syntax as Haskell. To prevent overwhelming you with the language's full grammar, the syntax will be shown by examples in the problems. You can also look at the parser to see how statements and expressions are formed.

REPL

Next is the REPL (and a main function which calls the REPL with empty environments). `repl` waits for a line of input, then calls the parser code on this input to convert it into a `Stmt`. It then proceeds to call `exec` on the `Stmt`, printing any result and updating the environments. This loops until you input `quit;`.

Problems

Your task is to implement the rest of the interpreter; in particular, the lifting functions, the expression evaluator, and the statement executor.

Lifting Functions

Since we are not directly interacting with Haskell's primitive values (but with values of type `Val` which represent our custom language's set of primitive values), we cannot directly apply Haskell's builtin functions either. For example, we cannot directly evaluate `IntVal 5 + IntVal 8`. However we do want a function can add two `IntVal`s and get back an `IntVal`. To do this, we could write a function like:

```haskell
intValAdd :: Val -> Val -> Val
intValAdd (IntVal x) (IntVal y) = IntVal (x + y)
```

This unpacks the two parameter `IntVal`s using pattern matching, adds them using the builtin `+`, and then puts the sum back into an `IntVal`. However, we have a lot of builtin primitive functions for our language, and writing a function for each of them would be tedious, and it would be more difficult to add primitive functions to the language.

Thus, we have the lifting functions which "lift" a builtin Haskell function to act on `Val`s instead. There are three lifting function for the relevant function types: `liftIntOp` lifts binary integer functions like `+`, `liftBoolOp` lifts binary boolean functions like `&&`, and `liftCompOp` lifts comparison functions like `<=`. Note that even though Haskell can compare booleans, we can only compare integers in our language. If the types of the values passed into a lifting function are incorrect you should return the exception `ExnVal "Cannot lift"`.

`liftIntOp` is written for you. You must write `liftBoolOp` and `liftCompOp`. Note that the output appears as if it were not a `Val` type because we provided a `Show` instance for `Val`.

```
*Main> let intValAdd = liftIntOp (+)
*Main> intValAdd (IntVal 5) (IntVal 4)
9
*Main> liftIntOp mod (IntVal 13) (IntVal 5)
3
*Main> liftBoolOp (&&) (BoolVal True) (BoolVal False)
False
*Main> liftBoolOp (||) (IntVal 1) (IntVal 0)
exn: Cannot lift
*Main> liftCompOp (<=) (IntVal 5) (IntVal 4)
False
*Main> liftCompOp (==) (BoolVal True) (BoolVal True)
exn: Cannot lift
```

Eval

The `eval :: Exp -> Env -> Val` function takes an expression and the current environment (the values stored in the variables in scope), and evaluates the expression given that environment to get a value. You will need to implement the `eval` function for each possible type of expression. The rules describing how to do so, the semantics, are in `semantics.pdf`. You can test the function by calling `eval` directly, or by calling `print` on an expression in the REPL.

Constants

Before we can do any evaluation, we'll need to define some basic expressions in `Exp` for `eval`. In particular, modify `eval` to handle both `IntExp`s and `BoolExp`s.

```
*Main> eval (IntExp 5) H.empty
5
*Main> eval (BoolExp True) H.empty
True
```

```
Welcome to your interpreter!
> print 5;
5
> print true;
True
> quit;
Bye!
```

Variables

Modify `eval` to handle `VarExp`s. (Notice that we have no way to add variables to the environment in the REPL yet, but we can call `repl` directly with a non-empty environment.)

```
*Main> let env = H.fromList [("x", IntVal 3), ("y", IntVal 5)]
*Main> eval (VarExp "x") H.empty
exn: No match in env
*Main> eval (VarExp "x") env
3
*Main> eval (VarExp "y") env
5
*Main> repl H.empty env [] ""
> print x;
3
> print x + y;
8
> print z;
exn: No match in env
```

Arithmetic

Modify `eval` to handle `IntOpExp` so that we can evaluate arithmetic expressions. Note that division must be handled specially to throw an exception in the case of a division by zero. `liftIntOp` will come in handy.

Note that for this (and the following problem) if we are using the REPL, the `String` representing the operator we want to apply will always be a valid operator that can be found in one of the primitive function maps - otherwise the expression would not have made it past the parser. Thus it is okay (for this assignment) to forgo handling a failed lookup for the operator. You'll notice there are semantic rules for this though, so you can handle failed lookup if you want to (it won't be tested).

```
*Main> eval (IntOpExp "+" (IntExp 5) (IntExp 4)) H.empty
9
*Main> eval (IntOpExp "-" (IntOpExp "*" (IntExp 3) (IntExp 10)) (IntExp 7)) H.empty
23
*Main> eval (IntOpExp "/" (IntExp 6) (IntExp 2)) H.empty
3
*Main> eval (IntOpExp "/" (IntExp 6) (IntExp 0)) H.empty
exn: Division by 0
*Main> eval (IntOpExp "+" (IntExp 6) (IntOpExp "/" (IntExp 4) (IntExp 0))) H.empty
exn: Cannot lift
```

```
Welcome to your interpreter!
> print 5 + 4;
9
> print (3 * 10) - 7;
23
> print 6 / 0;
exn: Division by 0
```

Boolean and Comparison Operators

Modify `eval` to handle `BoolOpExp` and `CompOpExp`.

```
*Main> eval (BoolOpExp "and" (BoolExp True) (BoolExp False)) H.empty
False
*Main> eval (CompOpExp "/=" (IntExp 4) (IntExp 6)) H.empty
True
```

```
Welcome to your interpreter!
> print true and true;
True
> print 3 < 4;
True
> print ((3 * 5) >= (20 - 6)) or false;
True
```

If Expressions

Modify `eval` to handle `IfExp`.

```
*Main> eval (IfExp (BoolExp True) (IntExp 5) (IntExp 10)) H.empty
5
*Main> eval (IfExp (BoolExp False) (IntExp 5) (IntExp 10)) H.empty
10
```

```
*Main> eval (IfExp (IntExp 1) (IntExp 5) (IntExp 10)) H.empty
exn: Condition is not a Bool
```

```
Welcome to your interpreter!
> print if true then 5 else 10 fi;
5
> print if false then 5 else 10 fi;
10
> print if false then 5 / 0 else 5 / 1 fi;
5
> print if 1 then 5 else 10 fi;
exn: Condition is not a Bool
```

Functions and Function Application

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. Then, modify `eval` to handle `AppExp`. This, in conjunction with `FunExp`, will allow us to do function application.

Note: You can assume for the purposes of this assignment that the number of arguments passed when a function is applied the same as the number that it needs.

```
*Main> let env = H.fromList [("x", IntVal 3)]
*Main> let fun1 = FunExp ["a", "b"] (IntOpExp "+" (VarExp "a") (VarExp "b"))
*Main> eval fun1 H.empty
<["a","b"], IntOpExp "+" (VarExp "a") (VarExp "b"), fromList []>
*Main> let fun2 = FunExp ["k"] (IntOpExp "*" (VarExp "k") (VarExp "x"))
*Main> eval fun2 env
<["k"], IntOpExp "*" (VarExp "k") (VarExp "x"), fromList [("x",3)]>
*Main> eval (AppExp fun1 [IntExp 5, IntExp 4]) H.empty
9
*Main> eval (AppExp fun2 [IntExp 5]) env
15
*Main> let envf = H.fromList [("f", eval fun1 H.empty), ("g", eval fun2 env)]
*Main> eval (AppExp (VarExp "g") [IntExp 4]) envf
12
*Main> eval (AppExp (IntExp 5) []) H.empty
exn: Apply to non-closure
```

```
Welcome to your interpreter!
> print fn [x] 2 * x end;
<["x"], IntOpExp "*" (IntExp 2) (VarExp "x"), fromList []>
> print apply fn [x] 2 * x end (4);
8
> print fn [a, b] if a <= b then a else b fi end;
```

```
<["a","b"], IfExp (CompOpExp "<=" (VarExp "a") (VarExp "b")) (VarExp "a") (VarExp "b"), fror
> print apply fn [a, b] if a <= b then a else b fi end (5, 7);
5
> print apply fn [a, b] if a <= b then a else b fi end (7, 5);
5
> print apply 5 ();
exn: Apply to non-closure
```

Let Expressions

Modify `eval` to handle `LetExp`.

```
*Main> eval (LetExp [] (IntExp 5)) H.empty
5
*Main> eval (LetExp [("x", IntExp 5)] (VarExp "x")) H.empty
5
*Main> eval (LetExp [("x", IntOpExp "+" (IntExp 5) (IntExp 4))] (IntOpExp "*" (VarExp "x") (
18
```

```
Welcome to your interpreter!
> print let [] 5 end;
5
> print let [x := 5] x end;
5
> print let [x := 5 + 4] x * 2 end;
18
> print let [x := 5] let [x := 6] x end end ;
6
> print let [x := 5] let [x := 6; y := x] y end end;
5
> print let [f := fn [a, b] a + b end] apply f(5,4) end;
9
> print let [x := 3] let [g := fn [k] k * x end] apply g(5) end end;
15
```

Statements

The `exec :: Stmt -> PEnv -> Env -> Result` function takes a statement and the current procedure and variable environments, and executes that statement in those environments to get its result. The result of executing a statement consists of an output string (possibly empty) and the possibly updated procedure and variable environments. You will need to handle the various kinds of statements. The rules describing the semantics are in `semantics.pdf`. You can test the function by calling `exec` directly, or by inputting the statements into the REPL. `PrintStmt` is done for you, and `QuitStmt` is already handled in `repl`.

Set Statements

Modify `exec` to handle `SetStmts`s. The output string should be empty.

```
*Main> exec (SetStmt "x" (IntExp 5)) H.empty H.empty
("",fromList [],fromList [("x",5)])
*Main> exec (SetStmt "y" (VarExp "x")) H.empty (H.fromList [("x", IntVal 5)])
("",fromList [],fromList [("x",5),("y",5)])
```

```
Welcome to your interpreter!
> x := 5;

> print x;
5
> do x := 7 ; print x; od;
7
> print x;
7
> do f := fn [a, b] a + b end; print apply f(5, 4); od;
9
```

Sequencing

Modify `exec` to handle `SeqStmts`. The output string should be the concatenation of all output strings of each individual statement.

```
*Main> exec (SeqStmt [PrintStmt (IntExp 5)]) H.empty H.empty
("5",fromList [],fromList [])
*Main> exec (SeqStmt [PrintStmt (IntExp 4), PrintStmt (IntExp 2)]) H.empty H.empty
("42",fromList [],fromList [])
```

```
Welcome to your interpreter!
> do print 6; od;
6
> do print 4; print 2; od;
42
> do print true + 5; print 7; print 7; od;
exn: Cannot lift77
```

If Statements

Modify `exec` to handle `IfStmts`. The output string depends on which statement is executed, and may also possibly be an error message.

```
*Main> exec (IfStmt (BoolExp True) (PrintStmt (IntExp 5)) (PrintStmt (IntExp 10))) H.empty H
("5",fromList [],fromList [])
*Main> exec (IfStmt (BoolExp False) (PrintStmt (IntExp 5)) (PrintStmt (IntExp 10))) H.empty
```

```
("10",fromList [],fromList [])
*Main> exec (IfStmt (IntExp 1) (PrintStmt (IntExp 5)) (PrintStmt (IntExp 10))) H.empty H.emp
("exn: Condition is not a Bool",fromList [],fromList [])
```

```
Welcome to your interpreter!
> if true then print 5; else print 10; fi
5
> if false then print 5; else print 10; fi
10
> if 4 < 3 then print true + 5; else do print 4; print 21; od; fi
421
> if 1 then print 5; else print 10; fi
exn: Condition is not a Bool
```

Procedure and Call Statements

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

Note: You can assume for the purposes of this assignment that the number of arguments passed into a procedure is the same as the number that it needs.

```
Welcome to your interpreter!
> procedure p() print 5; endproc

> call p();
5
> call q();
Procedure q undefined
> do procedure f(a, b) print a + b; endproc call f(5, 4); od;
9
> do procedure s(v) x:= v; endproc call s(10); print x; od;
10
> do procedure e(x) print true; endproc call e(23); print x; od;
True23
> do y := 0; procedure c() if y < 10 then do print y; y := y + 1; call c(); od; else print t
0123456789True
> do procedure fog(f, g, x) x := apply f(apply g(x)); endproc call fog(fn [x] x * 2 end, fn
14
```