FINAL CODE PROJECT: CS 4450

DUE: 3:00PM, 12/16/17

Directions

* The same general directions from Homeworks 1–3 for naming your submissions.
* When you submit, you must indicate which problems you attempted. Note that solutions to later problems depend on earlier problems. See the specifications for details.
* You must make a good faith attempt to answer at least the first problem.
* Your base grade for this assignment starts at 75% and increases based on the following factors:
  (a) Quality of submitted code (style will help, but not hurt your grade),
  (b) Complete coverage of randomly generated test cases (automatic full credit),
  (c) The total number of problems attempted and completed.
* This project uses the following file (available for download on the course canvas site):`Base.hs, Lexer.hs, Operators.hs, RecursiveFunctions.hs, RecursiveFunctionsAST.hs, RecursiveFunctionsParse.hs`.
* Every purported solution should be accompanied by tests written in HSpec.

Problems

1. Extend the definition of the `eval` function to handle multiple declarations. Notice first that the syntax of local declarations has changed (in `RecursiveFunctionsAST.hs`):

```
data Exp = Literal    Value
         | Unary      UnaryOp Exp
         | Binary     BinaryOp Exp Exp
         | If         Exp Exp Exp
         | Variable   String
         | Declare    [(String,Exp)] Exp -- This has changed.
         | RecDeclare String Exp Exp
         | Function   String Exp
         | Call       Exp Exp
        deriving (Eq, Show)
```

Now, a declaration can take multiple bindings, so it will generally have the form:

$$\texttt{Declare [}(string_1, exp_1), \ldots, (string_n, exp_n)\texttt{]}\ body$$

A call to the evaluation function, `eval Declare [`$(string_1, exp_1), \ldots, (string_n, exp_n)$`]` *body* `env`, should process these declarations in the following way:

1. Apply `eval` to each of $exp_1, \ldots, exp_n$ in the environment `env`, producing values $value_1, \ldots, value_n$,
2. Make a new environment adding bindings $(string_1, value_1), \ldots, (string_n, value_n)$ to `env`;
3. Apply `eval` to *body* in this new environment.

You can then test your answer on the expressions, `exp1,...,exp4`, defined in the project template. Note that `exp4` contains a free occurrence of the variable m (because `eval` uses Scope Rule 2, see below); and as such, it should crash. You can also come up with your own test cases.

2. This problem concerns *free variables*. Recall from lecture that an occurrence of a variable in an expression is *free* if it does not occur within the scope of a declaration of that variable. Consider the following example written in concrete syntax:

$$var\ m = 42, n = \underset{1}{m}; \underset{2}{n} + \underset{3}{m};$$

Three variables occur in this expression; labeled 1, 2, and 3. Occurrences 2 and 3 are bound by the declarations $n = m$ and $m = 5$, respectively. But what of the first occurrence? Well, that depends on the scope rules we pick; and thus, occurrence 1 could be either *free* or *bound*.

Let's define two scope rules. To do so, consider a declaration expression of the form: $var\ x_1 = e_1, \ldots, x_n = e_n;$ *body*.

Rule 1. The scope of declaration $x_i = e_i$ includes $e_{i+1}, \ldots, e_n$ and *body*. As such, given the declaration $x_1 = e_1$, x_1 does not occur free in $e_2, \ldots, e_n$ or *body*. With this rule, occurrence 1 is bound by the declaration $m = 5$.

Rule 2. The scope of each declaration, $x_1 = e_1, \ldots, x_n = e_n$, consists only of *body*. In particular, the scope of the declarations of $x_i = e_i$ do NOT include the $e_1, \ldots, e_n$. By this rule, occurrence 1 above is free, because it does not occur within the scope of $m = 5$ – since occurrence 1 does not occur in *body*.

Write two functions, `freeByRule1 :: [String] -> Exp -> [String]`, that finds the free occurrences of an expression according to scope defined by Rule 1 and `freeByRule2 :: [String] -> Exp -> [String]`, that finds the free occurrences of an expression according to scope defined by Rule 2. The cases that are most important to think through are the cases where variable declarations occur: `Declare, RecDeclare, and Function`. These functions must be written in accumulator passing style. In the specified type signatures, the first argument – namely, `[String]` – is a list of variables that have already been declared.

3. This problem implements a command-line *REPL* (Read-Eval-Print Loop). The template contains code for a simple REPL. Go ahead and try it out. You can type in expressions, they are evaluated, the value is printed, and the loop starts over. you can quit by typing "quit" at the prompt.

Implement a REPL using the Haskeline library. Your interface should have history features, and it should be able to process expressions typed directly into the prompt, as well as, expressions written in files. Unlike the simple REPL, your interface should test expressions for free variables using Scope Rule 2; and, instead of crashing, it should handle free variables by notifying the user with a well-formated message. If anyone decides to do this, I will give you some utility functions to access file contents.

Grading

Function	**Points**
`eval`:	+4%
`freeByRule1`:	+4%
`freeByRule2`:	+4%
`REPL`:	+9%
`Hspec Tests`:	max +4%
Total	+25%