Computer Science 320 (Fall 2016)
Concepts of Programming Languages

Problem Set 6:
Call-by-value Interpreter

Out: November 10, 2016
Due: Monday, November 21, 2016, by 11:59 pm

In this assignment, you will implement a call-by-value interpreter for a small subset of Haskell. You will make changes to the `Env` and `Eval` modules.

Included with the code is a parser library, *Parsec*, necessary to compile and use the interpreter. When using Hugs, it should be sufficient to load the `Main` module, and to run `mainParseEval "tests1.mhs"`. The files should run as provided, so let us know if you run into problems.

Familiarize yourselves with the `Exp` data structure in the `Exp` module (`Exp.hs`), which represents the abstract syntax of the language. The `Val` data structure, found in the `Val` module (`Val.hs`), represents evaluated (a.k.a. normalized) expressions in the language. Note that the two correspond in the base cases (integers, booleans, unit, empty list, built-in operators, and lambda abstractions).

Note that you are not allowed to modify any of the existing data type definitions unless a problem states otherwise or explicitly directs you to change a definition. There are, however, crude `show` functions for expressions, and you may want to use them at times when debugging (think about returning a subexpressions as a string in an error message).

Problem 1. (30 pts)

In this problem, you will implement an interpreter for expressions without variables, abstractions, or let-bindings. Because the parsed code is not being type checked, errors may occur during evaluation, so you will need to use `Error Val` as the return type for the various evaluation functions.

(a) In the `Eval` module, implement the body of the function `appOp :: Oper -> Val -> Error Val`, which takes an operator and a value, and when the operator is defined on that value in mathematical terms, returns the result (unary operators include `Not`, `Head`, and `Tail`.

If the operator is binary, take advantage of the `Partial::Oper -> Val -> Val` constructor, which can represent a partially applied built-in function. For example, `appOp` might return a value like `Partial Plus (N 4)` if it must apply the binary operator `Plus` to the single value `N 4`.

If the operator is not defined on that value, the function should return an error. Be careful with list operators such as `Head`, as they are not defined on empty lists. When in doubt, you can use the real Haskell interpreter to determine how this function should behave.

(b) In the `Eval` module, implement the body of the `appBinOp::Oper -> Val -> Val -> Error Val` function, which takes an operator and two values, and returns the resulting value if it is defined. Otherwise, it returns an error. Don't forget that `Equal` is defined on both numbers and booleans. You are not required to implement equality on lists, but you may do so for a small amount of extra credit.

(c) In the `Eval` module, implement the body of the `appVals::Val -> Val -> Error Val` function, which takes a pair of values where the first value is either a unary operator, a binary operator, or a partially applied binary operator. You should not need to use any functions other than those you defined in parts (a) and (b).

(d) In the `Eval` module, implement the body of the `ev0::Exp -> Error Val` function, which evaluates expressions which do not contain let-bindings, lambda abstractions, or variables. Thus, it should evaluate all base cases (such as operators, booleans, integers, unit, and the empty list) as well as `if` statements and applications. For all other cases, `ev0` may return an error.

You *should not* evaluate both branches of an `if` statement, and this is tested in the file `tests1.mhs`.

You should now be able to test the interpreter on some input. When using Hugs, it should be sufficient to load the `Main` module, and to run `mainParseEval "tests1.mhs"`. You may, of course, write and try evaluating your own test code.

Problem 2. (20 pts)

In the module `Env`, you will implement an environment data structure which can be used to store the values (and in later assignments, types) of variables.

(a) We will represent variable names with values of type `String`. Choose a representation for environments by choosing a definition for the polymorphic type `Env a`, which is used to store relationships between variable names and values of type `a`. Make sure the type name is exported from the module, as it will be needed in the `Eval` module.

(b) Define a value `emptyEnv::Env a` which will represent the empty environment (no variables), and make sure it is exported from the `Env` module.

(c) Define a function `updEnv::String -> a -> Env a -> Env a` which updates the environment with a new association between a variable and a value. If an association already exists in the environment, it should ***not*** be removed or overwritten. The new association should, however, hide any previous associations from being found. Make sure this function is exported from the module.

(d) Define a function `findEnv::String -> Env a -> Maybe a` which takes a variable name and retrieves the value associated with that variable, returning `Nothing` if there exists no such association in the environment. If in a given environment, multiple associations exist with the variable name in question, this function should always return the associated value which was inserted most recently. Export this function from the module.

Problem 3. (50 pts)

You will now implement a full interpreter for the language.

(a) In the `Eval` module, add two more cases to the body of the `appVals::Val -> Val -> Error Val` function, one for lambda abstractions with variables, and one for lambda abstractions with unit. You will need to call the `ev::Exp -> Env Val -> Error Val` function, which you have not yet implemented.

Note that a lambda abstraction with unit can only be applied to the unit value. Note also that a lambda abstraction is represented at the value level as a *closure* – it holds a copy of the environment under which the abstraction occurred in the program. The expression inside the lambda abstraction should be evaluated under this stored environment. For non-unit lambda abstraction, you should remember to extend this environment, however, by associating the formal variable in the abstraction with the argument value.

(b) Implement the body of the `ev::Exp -> Env Val -> Error Val` function, which evaluates all expressions which do not produce an error, and returns an error otherwise.

The base cases should be similar to those of `ev0`, except for the case of a variable. A variable evaluates to the value with which it is associated in the environment. If it is not found in the environment, it is not bound, and an error should be returned. When encountering lambda abstractions, remember to store the current environment inside them.

Remember that this interpreter is call-by-value, so `let` bindings and arguments should be evaluated eagerly. And, as before in `ev0`, you *should not* evaluate both branches of an `if` statement.

(c) Modify the `evalExp::Exp -> Error Val` function in the `Eval` module to call `ev` instead of `ev0`.

You should now be able to test your interpreter. The file `tests2.mhs` contains a program which outputs the prime numbers between 2 and 20. You are encouraged to write additional test cases, and particularly insightful or revealing test cases may receive some extra credit.