Computer Science 320 (Fall 2016)
Concepts of Programming Languages

Problem Set 7:
Call-by-name Interpreter

Out: Thursday, November 17, 2016
Due: Monday, December 5, 2016, by 11:59 pm

In this assignment, you will implement a call-by-name interpreter for a small subset of Haskell. You will continue to modify the Haskell skeleton code from Assignment #6. You may use the solutions for Assignment #6, either those posted or your own, as a starting point. You will make changes to the `Exp` and `Eval` modules.

Note that you are still not allowed to modify any of the existing data type definitions unless a problem states otherwise or explicitly directs you to change a definition.

Note: Problem 4 is a bonus problem, a solution for which is worth up to 25 points of extra credit.

Problem 1. (30 pts)

(a) In the `Exp` module, implement a recursive function `noLets::Exp -> Exp` which takes an expression as an argument, and converts all `let` subexpressions found in that expression into applications of lambda abstractions.

(b) In the `Exp` module, implement a function `subst::String -> Exp -> Exp -> Exp` which takes a variable name x, an expression N for which that variable must be substituted, and finally, an expression M on which to perform the substitution.

The function can perform the substitution naively (that is, you may assume that there are no variable name collisions), but it must not substitute any bound variables, even if they have the same name. Think carefully about the checks which need to be performed when `subst` encounters a lambda abstraction.

If throughout the assignment you never call `subst` on an expression which may have a `let` binding as a subexpression, you do not need to include a case for `let` bindings in your definition of `subst`.

Problem 2. (45 pts)

(a) In the `Eval` module, implement a function `appValExp::Val -> Exp -> Error Val` which evaluates a value applied to an expression. You should not evaluate the second arguments of the short-circuited boolean binary operators (i.e. `((&&) False)` and `((||) True)`), nor the argument passed to a unit lambda (that is, `\() -> ...`) abstraction. Note that for some cases, `appValExp` will need to call `subst`, as well as `ev0`, which you will implement in part (b). For convenience, you may call `appVals` from the previous assignment, but do this very carefully. You should not evaluate any subexpression which does not need to be evaluated according to the evaluation rules.

(b) In the `Eval` module, modify the body of the `ev0::Exp -> Error Val` function so that it evaluates *all possible* expressions according to the call-by-name evaluation *substitution* model. If you apply `noLets` to an expression before calling `ev0` (for example, by modifying the wrapper `evalExp::Exp -> Error Val`), you can be certain that the only situation in which you will need to perform a substitution is at an application of a lambda abstraction, which is a case already handled by `appValExp`.

Problem 3. (25 pts)

(a) In the `Eval` module, modify the body of the `ev::Exp -> Env Val -> Error Val` function so that it evaluates expressions according to the call-by-name evaluation *environment* model. You may use the `subst` function to replace variables with thunks applied to unit. In your solution, you may simply reuse the variable being substituted as the variable bound to the thunk.

(b) Create a file named `tests3.mhs`, and in it, write a small program on which the call-by-value interpreter implemented in the last assignment would diverge, but on which the call-by-name interpreter implemented in this assignment converges.

Problem 4. (*25 extra credit pts)

You will implement a transformation on expressions which ensures that the implementation of `subst` in Problem 1(b) works correctly for all programs.

(a) In the `Exp` module, implement a recursive function `unique::[String] -> Env String -> Exp -> (Exp, [String])` which takes an expression and gives a unique name to every variable in that expression, ensuring that no two lambda abstractions are over variables with identical names. You will need to maintain a list of fresh variable names. This list must also be returned with the result because an expression may have multiple branches, and variables cannot repeat even across branches. The environment is used to associate old variable names with new ones. You may use `noLets`.

(b) Modify `evalExp::Exp -> Error Val` so that an expression is evaluated only after being transformed using `unique`.