Exercise Sheet Seven : Interpreters

The aim of this tutorial is for you to build an interpreter for the lambda-calculus by using a machine based implementation. By the end of the exercises you should be familiar with the CEK machine model and how to vary it to include new features.

Attempt the exercsise in your own time before Tuesday's lab and use that session for obtaining feedback and for asking questions related to the exercises.

Throughout this sheet we will be working with the data type for lambda calculus abstract syntax trees given in the lecture

```
data Expr = Var String | Lam String Expr | App Expr Expr
   deriving (Eq,Show,Read)
```

We are going to implement the CEK machine for Call-by-Value lambda calculus as presented in the lecture. In order to check your implementation you may like to use my substitution based lambda calculus interpreter as demonstrated in the lectures.

Exercise One

First define suitable types for environments and extend the Expr datatype with closure expressions. Recall that closures comprise a pair of a lambda expression and an environment, and environments contain mappings from variable names to closures.

Having done this, define a `lookup` function that, given a string name of a variable and an environment, returns the closure that the variable is bound to in that environment. Think about what should you return in case the variable has no binding in the environment.

Exercise Two

Define suitable types for frames and continuations. Given this we can then define a type for what we shall call a *configuration*. A configuration is the triple consisting of the current Control, Environment, and Continuation.

Exercise Three

Implement a function called `eval1` that takes a configuration and returns the configuration given by executing one step of the machine by applying the rules R1-R5 given in the lecture notes. Pattern matching is very much your friend when it comes to defining this function. You will find it helpful to define some additional functions for manipulating closures and environments also.

Exercise Four

We should be able to put this all together now to implement a function called `eval` that takes a lambda expression and evaluates it using a Call-by-Value strategy until termination. The function should return the terminated expression (if it terminates). You will need to think what the initial configuration should be and how to recognise termination.

Exercise Five

An example lambda calculus term is given by

```
App (Lam "v"
     (App
       (App (Var "v") (Lam "z" (Var "z")))
       (App (Lam "v" (App
                         (App (Var "v") (Lam "x" (Lam "y" (Var "x"))))
                         (Lam "z" (Var "z")))
             )
             (Lam "x" (Lam "y" (Var "x")))
       )
     )
    )
(Lam "x" (Lam "y" (Var "y")))
```

It should evaluate to `Lam "x" (Lam "y" (Var "x"))`. Test your implementation on this term to check that it does. Build some other simple terms to test with.

Exercise Six

This exercise is a little harder and is for those who like a challenge.

Extend the lambda calculus with natural numbers by providing terms `Zero` and `Succ e` to represent natural numbers. Do this by modifiying the data type above for `Expr` above to include `Zero` and `Succ Expr`. Extend the CEK machine implemenation above to cater for this extended langauge. Hint: you will need to identify what the terminated values of the language are, modify frames accordingly and rewrite the machine rules R1-R5.