Exercise 06

Task 1: About Recursion

We saw in the lecture that to implement recursion in an interpreter with environments and closures, one needs to create a cyclic environment which when we look up a recursive function will return itself again.

1. Consider the interpreter for RCFLAE with immediate substitution. How is recursion handled here? In comparison, why do we need to deal with creating a cyclic environment when implementing an interpreter with environments?

Answer: With immediate substituion, we can simply, whenever we encounter a `Rec` construct during interpretation, look in the body and replace any self-references with the `Rec` expression we are interpreting, thereby generating recursion. If we want to implement the more efficient interpreter that uses environments, however, we need to have the necessary cyclic dependency in the environment: The environment needs to be able to produce as many bindings of the recursive function as necessary for executing any program.

We saw in the lecture that you can use various features of the *interpreter* language to implement recursion in the *interpreted*language. For example, the of RCFLAE with environments made use of Scala's mutable maps to achieve the necessary cyclic binding structure for a recursive function. Later in the lecture, you saw another interpreter for RCFLAE that used Scala's functions as environments and Scala's lazy values to achieve a "call-by-need" evaluation of environments with recursive bindings.

In our RCFLAE interpreter, we used Scala's mutation to realize cyclic environments. In particular, we used `collection.mutable.Map`:

```
case Rec(boundId, namedExpr, boundBody) => {
  val recEnv = collection.mutable.Map() ++ env
  recEnv += boundId -> interp(namedExpr, recEnv)
  interp(boundBody, recEnv)
}
```

In SCFLAE, we augmented our language with state. We can even use this state to implement recursion.

1. Implement an interpreter RSCFLAE that is like SCFLAE but supports recursion. You are not allowed to use Scala's support for mutation (var, mutable.Map) but must use our language's support directly.

Task 2: Algebraic data types

Algebraic data types (ADTs) is a composite type used for representing structured data. ADTs are supported in many programming languages that feature functional programming. For example, in Haskell an ADT is defined as:

```
data Exp = Num Integer
         | Plus Exp Exp
         | Id String
         | Fun String Exp
         | App Exp Exp
```

Here, `Exp` is the name of the data type. `Num`, `Plus`, `Id`, `Fun`, and `App` are constructors of the data type `Exp`. They can be used like functions and when called yield a value of type `Exp`. The types occurring after the constructor names are the parameter types of the constructor. We use the constructors of an ADT to construct new values of that ADT, for example:

```
App (Fun "x" (Id "x")) (Num 17) :: Exp
```

To decompose an ADT value, we use pattern matching (again Haskell syntax):

```
interp env exp =
  case exp of
    Num n => NumV n
    Plus e1 e2 => (interp env e1) + (interp env e2)
    Id x => lookup env x
    Fun x body => ClosureV env x body
    App e1 e2 =>
      case interp env e1 of
        ClosureV env x body =>
          let arg = interp env e2
          in interp ((x,arg) : env) body
        _ => error("can only apply function values")
```

In Scala, ADTs are represented as case classes:

```
abstract class Exp
case class Num(n: Integer) extends Exp
case class Plus(e1: Exp, e2: Exp) extends Exp
...
```

with constructors and pattern matching:

```
Plus(Num(1), Num(2)) match {
  case Num(n) => n
  case Plus(e1, e2) => eval(e1) + eval(e2)
}
```

1. Add support for ADTs to FLAE (no state). It should be possible to write the following program in your language:

   data Nat = Zero | Succ Nat

   data Bool = False | True

   data NatList = Nil | Cons Nat NatList

filterNatList :: (Nat-> Bool) -> NatList -> NatList filterNatList f l = case l of Nil => Nil Cons x xs => if f x then x : (filterNatList f xs) else filterNatList f xs