Assignment 7(Interpreters)

This problem came from Geoffrey Smith

In this problem, you will develop an SML interpreter. The syntax is given by the following BNF grammar:

```
e ::= x | n
        | true
        | false
        | succ
        | pred
        | iszero
        | if e then e else e
        | fn x = e
        | e e
        | (e)
```

We will use the following datatype as the AST representing this grammar:

```
datatype term = AST_ID of string
              | AST_NUM of int
              | AST_TRUE
              | AST_FALSE
              | AST_SUCC
              | AST_PRED
              | AST_ISZERO
              | AST_IF of term * term * term
              | AST_FUN of string * term
              | AST_APP of term * term
```

You are to write an SML function interp that takes an abstract syntax tree, represented as a term, as well as an environment, represented as an env, and returns the result of evaluating it. Initially, we will define our result datatype as follows:

```
datatype result = RES_ERROR of string
                | RES_NUM of int
                | RES_TRUE
                | RES_FALSE
                | RES_SUCC
                | RES_PRED
                | RES_ISZERO
                | RES_FUN of (string * term);
```

The evaluation should be done according to the rules given below. (Rules in this style are known in the research literature as a *natural semantics*.) The rules are based on *judgments* of the form

```
env |- e = v
```

which means that term *e* evaluates to value *v* (and then can be evaluated no further). For the sake of readability, we describe the rules below using the concrete syntax of our language; remember that your interp program will actually need to work on abstract syntax trees, which are SML values of type term. In the rules below, each judgement will occur in the context of an environment. As before, you will be provided with an environment implementation, so you won't have to write one yourself.

Rules The first few rules are uninteresting; they just say that basic values evaluate to themselves:

```
(1) env |- n = n, for any non-negative integer literal n
(2) env |- true = true and env |- false = false
(3) env |- succ = succ, env |- pred = pred, and env |- iszero = iszero.
```

The interesting evaluation rules are a bit more complicated, because they involve *hypotheses* as well as a *conclusion.* For example, here's one of the rules for evaluating an if-then-else:

```
     env |- b = true         env |- e1 = v
(4) -----------------------------------------
         env |- if b then e1 else e2 = v
```

In such a rule, the judgments above the horizontal line are *hypotheses* and the judgment below is the *conclusion.* We read the rule from the bottom up: "if the expression is an if-then-else with components b, e1, and e2, and b evaluates to true and e1 evaluates to v, then the entire expression evaluates to v". Of course, we also have the symmetric rule:

```
    env |- b = false          env |- e2 = v
(5) ------------------------------------------
     env |- if b then e1 else e2 = v
```

The following rules define the behavior of the built-in functions:

```
      env |- e1 = succ        env |- e2 = n
(6)  ------------------------------------------
      env |- e1 e2 = n+1

      env |- e1 = pred        env |- e2 = 0
(7)   -----------------------------------------
       env |- e1 e2 = 0

       env |- e1 = pred        env |- e2 = n+1
      -------------------------------------------
           env |- e1 e2 = n

       env |- e1 = iszero   env |- e2 = 0
(8)   ---------------------------------------
         env |- e1 e2 = true

         env |- e1 = iszero   env |- e2 = n+1
    -----------------------------------------
        env |- e1 e2 = false
```

(In these rules, n stands for a non-negative integer.) For example, to evaluate

```
if iszero 0 then 1 else 2
```

we must, by rules (4) and (5), first evaluate iszero 0. By rule (8) (and rules (3) and (1)), this evaluates to true. Finally, by rule (4) (and rule (1)), the whole program evalutes to 1.The following rule describes variable evaluation

```
                 env(x) = v
(9)     ----------------------------
              env |- id x = v
```

Just like the built-in functions (succ, pred, and iszero), functions defined using fn evaluate to themselves:

```
(10)    env |- (fn x = e) = (fn x = e)
```

Computations occur when you *apply* these functions to arguments. The following rule defines *call-by-value* (or *eager*) function application, also used by SML: if the function is of this form

```
env |- fn x = e
```

Evaluate the operand to a value v1, and then evaluate the body in an extended environment, where v1 is bound to x.

```
    env |- e1 = (fn x = e)  env |- e2 = v1  env[x := v1] |- e = v
(11) ------------------------------------------------
                env |- e1 e2 = v
```

Similarly, the last rule implements let.

```
    env |- e1 = v1     env[x := v1] |- e = v
(12)    --------------------------------------------
        env |- let x = e1 in e end  = v
```

1 Problem 1

First, convince yourself that the above rules give semantics to a dynamically scoped call-by-value language. Then, using the above rules write an interpreter,

```
interp: (env * term) ->  result
```

2 Problem 2

Your next step is to implement a call-by-value statically scoped language. To that end, let us change the semantics of functions as follows:

```
(10a)    env |- (fn x = e) = ((fn x = e), env)
```

Notice that we save the environment in the return value. This environment will be used when we invoke the function:

```
 env |- e1 = ((fn x = e), env1)  env |- e2 = v1 env1, (x, v1) |- e = v
(11a) ---------------------------------------------------------
          env |- e1 e2 = v
```

As explained in the above rules, in order to achieve static scoping, you will need to add closures to our language. You can do this by adding a RES_CLOSURE tag to our result type, which contains both a function and an environment. Write a new statically scoped interpreter

```
 interp_static: (env * term) -> result
```

You should be able to copy a lot of the code from interp. NOTE: In constructing your closure, you may need to use mutually recursive datatype definitions. While SML generally requires datatype definitions to come before their uses, you can define two datatypes at the same time using the "and" keyword. For example:

```
datatype foo = FOO of int  | FOOBAR of bar
and      bar = BAR of bool | BARFOO of foo
```

3 Problem 3

The interpreters you have written so far implement call-by-value parameter passing, in which variables are evaluated as soon as they are defined. This results in eager behavior, as you've seen in SML. To implement call-by-name, in which variables are not evaluated until they are needed (which results in "lazy" behavior) we need to modify our rules:

1. For dynamic scope, we have:

```
             env(x) = e    env |- e = v
(9a) -----------------------------
            env |- x = v

        env |- e1 = (fn x = e3)  env, (x, e2) |- e3 = v3
(11b) -------------------------------------------------
           env |- e1 e2 = v3
```

2. For static scope, we have:

```
         env(x) = (e, env1)   env1 |- e = v
(9b) -----------------------------------
              env |- x = v

(10b)  env |- (fn x = e) = (fn x = e, env)

        env |- e1 = (fn x = e3, env1)  env1, (x, (e2, env)) |- e3 = v3
(11b) -------------------------------------------------
           env |- e1 e2 = v3
```

1. Write a new dynamically scoped interpreter

```
interp_lazy: (env * term) -> result
```

which implements call-by-name parameter passing, with dynamic scope.

2. Write a new statically scoped interpreter

```
interp_lazy_static (env * term) -> result
```

which implements call-by-name parameter passing with static scope.

You may need to modify some of your basic datatypes.

4 Problem 4

Briefly discuss what you would change in your interpreters if the input to the interpreter had been type-checked before its invocation.