Assignment 4 (Interpreters)

In this assignment, you will develop an SML interpreter for a small functional language called PCF, which stands for Programming language for Computable Functions. The language is relatively simple, yet powerful, with arithmetic expressions and functions. The syntax of PCF programs 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) | let x = e in e
```

In the above, `x` stands for an identifier; `n` stands for a non-negative integer literal; `true` and `false` are the boolean literals; `succ` and `pred` are unary functions that add 1 and subtract 1 from their input, respectively; `iszero` is a unary function that returns true if its argument is 0 and false otherwise; `if e1 then e2 else e3` is a conditional expression; `fn x => e` is a function with parameter `x` and body `e`; `e e` is a function application; `(e)` allows parentheses to be used to control grouping; and `let x=e1 in e2` is a let expression.

It should be clear to you that the above grammar is quite ambiguous. For example, should `fn f => f f` be parsed as `fn f => (f f)` or as `(fn f => f) f`? We can resolve such ambiguities by adopting the following conventions (which are the same as in SML):

- Function application associates to the left. For example, `e f g` is `(e f) g`, not `e (f g)`.
- Function application binds tighter than `if`, and `fn`. For example, `fn f => f 0` is `fn f => (f 0)`, not `(fn f => f) 0`.

We don't want to interpret concrete syntax directly. Instead, the interpreter will work on a parse tree (also called abstract syntax tree) representation of the program; these syntax trees will be values of the following SML datatype:

```
datatype term = AST_ID of string | AST_NUM of int  | AST_BOOL of bool
              | AST_SUCC | AST_PRED | AST_ISZERO | AST_IF of term * term * term
              | AST_FUN of string * term | AST_APP of term * term
              | AST_LET of (string * term * term)
              | AST_ERROR of string
```

This definition mirrors the BNF grammar given above; for instance, the constructor `AST_ID` makes a string into an identifier, and the constructor `AST_FUN` makes a string representing the formal parameter and a term representing the body into a function. Note that there is no abstract syntax for `(e)`; the parentheses are just used to control grouping.

Practice[1]

Write the PCF abstract syntax tree term corresponding to the following PCF source expressions (do this manually so that you understand; later, use your PCF parser to confirm your answers).

[1] You do not have to submit anything for this part

- (a) `123`
- (b) `(fn x => x ) 123`
- (c) `if iszero x then x else 0`

Continuing

You will be building an interpreter that will evaluate Abstract Syntax Trees (ASTs) for PCF to a final AST that represents a computed result.

Instead of building the parse trees for arithmetic expressions by hand (as in the previous problem), we are providing you with a parser that converts from concrete PCF syntax to an abstract syntax tree. The parser is available parser.sml. Include the command

```
use "parser.sml";
```

at the beginning of the file containing your interpreter. The parser defines the datatype term as well as two useful functions, `parsestr` and `parsefile`. Function `parsestr` takes a string and returns the corresponding abstract syntax; for example

```
- parsestr "iszero (succ 7)";
val it = AST_APP (AST_ISZERO,AST_APP (AST_SUCC,AST_NUM 7)) : term
```

Function `parsefile` takes instead the name of a file and parses its contents. (By the way, the parser is a recursive-descent parser; you may find it interesting to study how it works.)

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, as a result. Initially, we will define our result datatype as follows:

```
datatype result = RES_ERROR of string | RES_ERROR of string | RES_NUM of int
                | RES_BOOL of bool | 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 PCF programs; remember that your `interp` program will actually need to work on SML values of type `term`.

Environments

Interpreters also need a way of passing parameters to user-defined functions. In our interpreter, we will be using environments. An environment is at its core, a set of relations, from names to values. Whenever a term is evaluated, it will be done in the context of an environment. This environment must be extendable to allow new variables to be bound, and it must be searchable, to allow bound variables to be later retrieved. For example, if we were to evaluate the expression:

```
let
    x = 4
in
    let
        y = 5
    in
        x+y
    end
end
```

We would start with an empty environment (). When we come to the first let expression, we would add the relation `(x,4)` to our environment, and evaluate the body of the let in that context. Similarly, when we come to the second let expression, we add `(y,5)` to our environment, and evaluate its body in the further extended environment. Thus, when we come to the expression `x+y`, we evaluate it in the environment `((x,4), (y,5))`. In order to determine the values of `x` and `y`, we merely look them up.

In the rules below, each judgement will occur in the context of an environment. For this assignment, 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 evaluates to 1.The following rule describes variable evaluation

```
                 env(x) = v (i.e. lookup x in env )
(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:

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

The above rule says that in order to evaluate an application `e1 e2` in an environment `env`, we first evaluate `e1` and `e2` in the same environment, if `e1` and `e2` evaluate to `fn x => e` and `v1` then the environment `env` is extended by binding variable `x` to `v1`, and in that extended environment one evaluates the body `e` of the function. If that returns `v` then `v` is the result of the application.

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
```

Do not assume that programs have been typed checked before, this means that you will need to return error terms for type errors like `succ true`. Here is a skeleton file (*interp.sml*) to help you to get started. It contains an implemented environment datatype and related functions, as well as a mostly unimplemented `interp` function. Once you've implemented your interpreter, you can test it with some of the examples in this file: interpExamples.sml.

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 by introducing the notion of a *closure* 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
(11)-------------------------------------------------------------------------
       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
```