Due: 21 February, 11:59:59 p.m.

In this assignment you will write an interpreter for a simple functional language called PCF. Your interpreter will be a SML function that takes two parameters, one which is an abstract syntax tree for the program, and the other is the environment of the program. Your solution should include several functions.

Language Definition. Your program wont deal directly with the PCF syntax, but it is used below. The grammar for the PCF language is

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

In the grammar "`x`" represents a variable, "`n`" represents an integer, "`true`" and "`false`" are Boolean values, and the remainder are function or expression values. The function "`succ`" adds one to its argument, "`pred`" subtracts one from its argument, and "`iszero`" returns "`true`" if its argument is 0 and "`false`" otherwise. The if-then-else is a conditional expression, "`fn x => e`" is a function with parameter "`x`" and body "`e`", "`(e e)`" is function application, and "`rec x => e`" is for defining recursive functions.

An *environment* is a binding of values to free variables in a PCF term. A *free variable*, or more precisely a *free* occurrence of a variable, `x` is an occurrence that does not occur as a function parameter. For instance, `x` is free in `(succ x)`, but is not free in `(fn x => (succ x))`. There are expressions in which a variable may have both free and nonfree occurrences such as `x` in `((fn x => (succ x)) (pred x))` An environment will be typically written as ρ, and the value of a variable x in the environment is written $\rho(x)$. When a binding of v to x is added to the environment ρ this is written as $\rho[x := v]$.

We will use a form of *natural semantics* to define the meaning of PCF expressions. The notation $(e, \rho) \downarrow v$ means that the expression e has the value v in the environment ρ. For example, if n is an integer then $(n, \rho) \downarrow n$. The meaning of compound expressions are defined by rules like

$$\frac{(\texttt{b}, \rho) \downarrow \mathit{true} \quad (e_1, \rho) \downarrow v}{(\texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2, \rho) \downarrow v}$$

which can be stated in English as

> The expression `if` b `then` e_1 `else` e_2 has value v in ρ if the expression b is true in ρ, and the expression e_1 has value v in ρ.

This notation is used in the following rules to define the semantics of PCF.

The values that are computed include integers, booleans, functions, thunks, and an error value. User defined functions will be represented by a *closure*, which includes the parameter name, the function body, and the environment in which the function was defined. A *thunk* is a way of suspending the evaluation of a term, and consists of the term and the environment in which it is defined.

The semantics of the PCF language are defined by the following rules. Let ρ be an environment.

1. $(n, \rho) \downarrow n$ for n an integer.
2. $(true, \rho) \downarrow true$, and similarly for false
3. $(x, \rho) \downarrow \rho(x)$, if $\rho(x)$ is not a thunk
4. $(x, \rho) \downarrow v$, if $\rho(x)$ is of the form $\texttt{thunk}(e, \rho')$ and $(e, \rho') \downarrow v$
5. $(error, \rho) \downarrow error$
6. $(\texttt{succ}, \rho) \downarrow succ$, and similarly for the other initial functions
7. $(\texttt{fn x => e}, \rho) \downarrow closure(x, e, \rho)$

   The value of a user defined function is a closure that includes the parameter, the body expression and the environment in which it is defined.
8. $$\frac{(\texttt{b}, \rho) \downarrow true \quad (\texttt{e}_1, \rho) \downarrow v}{(\texttt{if b then e}_1 \texttt{ else e}_2, \rho) \downarrow v}$$
9. $$\frac{(\texttt{b}, \rho) \downarrow false \quad (\texttt{e}_2, \rho) \downarrow v}{(\texttt{if b then e}_1 \texttt{ else e}_2, \rho) \downarrow v}$$
10. $$\frac{(\texttt{e}_1, \rho) \downarrow succ \quad (\texttt{e}_2, \rho) \downarrow n}{((\texttt{e}_1\ \texttt{e}_2), \rho) \downarrow (n+1)}$$
11. $$\frac{(\texttt{e}_1, \rho) \downarrow pred \quad (\texttt{e}_2, \rho) \downarrow 0}{((\texttt{e}_1\ \texttt{e}_2), \rho) \downarrow 0}$$
12. $$\frac{(\texttt{e}_1, \rho) \downarrow pred \quad (\texttt{e}_2, \rho) \downarrow n+1}{((\texttt{e}_1\ \texttt{e}_2), \rho) \downarrow n}$$
13. $$\frac{(\texttt{e}_1, \rho) \downarrow iszero \quad (\texttt{e}_2, \rho) \downarrow 0}{((\texttt{e}_1\ \texttt{e}_2), \rho) \downarrow true}$$
14. $$\frac{(\texttt{e}_1, \rho) \downarrow iszero \quad (\texttt{e}_2, \rho) \downarrow n+1}{((\texttt{e}_1\ \texttt{e}_2), \rho) \downarrow false}$$

15.

$$\frac{(\texttt{e}_1, \rho) \downarrow \mathit{closure}(x, \texttt{e}_3, \rho_f) \quad (\texttt{e}_2, \rho) \downarrow v_1 \quad (\texttt{e}_3, \rho_f[x := v_1]) \downarrow v}{((\texttt{e}_1\ \ \texttt{e}_2), \rho) \downarrow v}$$

The body of the function is interpreted in the environment from the closure, updated to reflect the assignment of the actual parameter value to the formal.

16.

$$\frac{(e, \rho[x := \texttt{thunk(rec x => e, } \rho\ \texttt{)}]) \downarrow v}{(\texttt{rec x => e}, \rho) \downarrow v}$$

When evaluating a recursive expression we evaluate the body in an environment in which the recursive name stands for the recursive expression. It is important to put this in the environment so that other recursive calls can be evaluated properly. We use thunks to suspend the evaluation of the call so that it will be not be evaluated until it is needed. As indicated in rule 3, when the thunk is encountered, the term is evaluated in the environment stored with it.

NOTE: If the expression does not match any of the rules, then the expression should evaluate to *error*.

Datatypes. You are to write an ML function `interp` that evaluates a term in a specified environment. The term will be represented as an abstract syntax tree (AST) in the following datatype `term`, and an environment by a value of the datatype `env` which is defined below.

```
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_ERROR |
   AST_FUN of (string * term) | AST_APP of (term * term) |
   AST_REC of (string * term);
```

The result of the evaluation will be an element of the ML datatype `value` defined as

```
datatype value = NUM of int | BOOL of bool | SUCC | PRED | ISZERO
               | CLOSURE of (string * term * env) | THUNK of term * env
               | ERROR
withtype env = string -> value;
```

The datatype `env` represents the type of environments. (The `withtype` construct a mutually recursive datatype definitions.) Notice that the environment type is just a function type.

When testing your function, it may be useful to use the parser that is in the file `pcfparser.sml` that is posted on the web. This file contains a parser that constructs an abstract syntax tree for an input program using the `term` datatype. You need to take care with the redefinition of datatypes, so see the note below about file structure to avoid problems when using this file.

Implementation.

1. Write an function to represent a default environment `default` that returns `ERROR` given any string `s`.

2. Define a higher-order function, `update(env,s,t)`, that returns an updated environment $env[s := t]$ (using the notation from above). In other words, when given an environment `env`, a string `s` representing a variable, and a value `t` returns a new environment that, when applied to `s` returns `t`, and otherwise returns what `env` would have returned.

   Remember that environments are functions. So, your function should take a function as a parameter, and return a function built from that function. While this seems like it should require fancy data structures, it is in fact a very simple function.

3. Define a function `interp(t,rho)` that takes as arguments a term `t`, and an environment `rho`, and returns a value representing the evaluation of `t` in the environment `rho`. The term `t` may include free variables, and the environment `rho` must have a binding for all the free variables in `t`. The initial environment should be the `default` function defined in part a.

Be certain to test your interpreter on the PCF for the following example written with some syntactic sugar:

```
let f = fn x => (iszero (succ x))
    in let g = fn y => f y
    in let f = fn x => (iszero x)
    in g 0
```

We can translate this to PCF using the correspondence that `(let x = M in N)` is `((fn x => N) M)`. This expression should evaluate to `false` under static scoping (the `f` in the definition of `g` is the outermost `f`), while it returns `true` under dynamic scoping (the `f` in the definition of `g` refers to the most recently defined value of `f` — the innermost one). Make sure your interpreter evaluates this expression properly (after you have replaced all of the let expressions).

Implementation Constraints. You are to use the datatypes given above. Your program should use only the pure functional features of SML. There is no reason to use references, input/output or other commands in this program. Only use type constraints if it is necessary to get a function to compile properly.

Commenting and Style. Your code should be neatly formatted with good indentation style. If you use emacs for editing, there is an sml-mode available from the SML/NJ website that will help with indentation. (You can get Xemacs for windows too.)

All functions must have block comments that explain what the function does and what the arguments are as well as giving the formal type of the function. Comment cases of functions when what they are doing is not clear. Your file must have a comment block that minimally identifies you as the programmer, lists your pid, and the functions that are implemented.

What to Submit. You are to submit a single (text) sml file to the curator containing your function definitions. The sml system should be able to read the function definitions in your file without needing any other files (i.e., there should be no `use` commands in your file).

Evaluation. Your program will be evaluated for correctness using automatic grading in the curator. The GTA will then look at your programs for programming style.

File Structure. You will be creating a single file that contains only your functions, but it needs the definitions of the datatypes to compile properly. These definitions are in the files `asttypes.sml` and `evaltypes.sml` that are posted on the web. I do not want you to include these definitions in your file either directly or by a `use` command. The reason is that my test program will load your file with a `use` command into a context in which those types are already defined. If your file redefines the datatypes, static scoping says that the two definitions are distinct. So, your functions will not accept ASTs from my functions. This will also cause problems if you want to use the `pcfparser.sml` file in which the types are already defined.

So, when you program you will need to first load either `asttypes.sml` or `pcfparser.sml`, and then load `evaltypes.sml` before loading your file. You might find it useful to write a little file that looks like

```
use "asttypes.sml"; (* or pcfparser.sml *)
use "evaltypes.sml";
use "mysolution.sml";
```

which you would load when you want to test your changes. If you are adventurous you could look into using `CM` the configuration management tool (which is like `make`, but is fancier).