Computer Science 131, Spring 2001
Assignment 7

Out: Thurday, March 22
Due: Wednesday, March 28, 2001

To complete this assignment, run SML/NJ on turing using `/cs/cs131/bin/sml-cs131-a7`. Put your answers in a file named `assign7.sml` and turn this in using `cs131submit`.

1 Environments and Lexical Scope

For this problem you are to modify the function `evalDyn` from the Assignment 5 to use *lexical* scope instead of dynamic scope. That is, you are to write a function

```
evalLex : (exp env) * exp -> exp
```

which can be compiled by the SML/NJ binary `/cs/cs131/bin/sml-cs131-a7`. (This is very like the `sml-cs131-a5` binary, with one change to the `exp` datatype, as described below.)

The defining characteristic of static scope is that the free variables of functions refer to those variables in scope *at the point where the function appears in the text of the program.* This can be modeled in an interpreter with an environment as follows: whenever we evaluate an expression of the form `fix f(x:t1):t2 is e`, we must grab a copy of the current environment (i.e., remember the values of all the free variables of this function). Then this copy of the environment is passed around *together* with the code of the function, so that when the function is finally applied we use this environment to find the values of the function's free variables (instead of the interpreter's "current" environment). This combination of code and an environment is called a *closure*.

For this problem, a closure will be represented as an expression of the form `Clo`(f, x, e, E); it contains two variables, an expression, and an environment.

The language must be changed so that a closure is a value but `fix f(x:t1):t2 is e` is not. Instead of evaluating to itself, a function definition will now evaluate to a closure.

$$
\begin{array}{rcl}
v & ::= & n \\
& | & \texttt{tt} \\
& | & \texttt{ff} \\
& | & \texttt{Clo}(f, x, e, E) \\
\\
e & ::= & v \\
& | & x \\
& | & e_1\texttt{+}e_2 \\
& | & e_1\texttt{<}e_2 \\
& | & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
& | & \texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2 \\
& | & e_1\, e_2 \\
& | & \langle e_1, e_2 \rangle \\
& | & \texttt{fst } e \\
& | & \texttt{snd } e \\
& | & e_1 \texttt{ :: } e_2 \\
& | & \texttt{case } e_1 \texttt{ of nil => } e_2 \texttt{ | } x\texttt{::}y \texttt{ => } e_3 \\
& | & \texttt{fix } f(x\texttt{:}t_1)\texttt{:}t_2 \texttt{ is } e
\end{array}
$$

The `exp` datatype that has been predefined for you is unchanged except for an additional case, used for representing closures:

```
datatype exp = ...
             | Closure of varname * varname * exp * exp env
```

Only three rules change in the dynamic semantics. (See Assignment 5 if you wish to refresh your memory on the others.) The first rule is written exactly the same as before, but has a slightly different meaning because the definition of values has changed. Closures evaluate to themselves, but the evaluation of functions requires a separate rule.

$$\frac{}{(E, v) \Downarrow v} \tag{1}$$

The new rule for evaluating a `fun` expression is:

$$\frac{}{(E, \texttt{fix } f(x\texttt{:}t_1)\texttt{:}t_2 \texttt{ is } e) \Downarrow \texttt{Clo}(f, x, e, E)} \tag{2}$$

That is, when we evaluate a function we get back a closure which contains not only enough information to execute the function but also the current environment (which contains all the values for the function's free variables).

Finally, the application rule must change.

$$\frac{\begin{array}{c}(E, e_1) \Downarrow (\texttt{Clo}(f, x, e_1', E_1)) \\ (E, e_2) \Downarrow v_2 \\ ((E_1, x\texttt{=}v_2, f\texttt{=}\texttt{Clo}(f, x, e_1', E_1)), e_1') \Downarrow v\end{array}}{(E, e_1\, e_2) \Downarrow v} \tag{3}$$

The first expression in the application will now evaluate to a closure. The key point to observe in this rule is that free variables in the function body e_1' have their values taken from E_1 (the environment stored in the closure) rather than from E (the environment when the function is called).

It will be probably easiest to write your `evalLex` function by starting with a definition of `evalDyn` function from Assignment 5, doing a search-and-replace to rename the function, making sure that closures evaluate to themselves, and modify the cases for `Fix` and `Apply`. Except for modifying the function's name, this is likely to require less than a dozen lines of code be changed.) It may be helpful to re-use test code from Assignment 5.

2 Abstracting Control.

The function

```
fold : ('a * 'b -> 'a) -> 'a -> 'b list -> 'a
```

is defined as follows:

```
fun fold f b []      = b
  | fold f b (x::xs) = fold f (f(b,x)) xs
```

So the code `fold f b [x1,...,x_n]` returns the result of computing

```
f(...(f(f(f(b,x1),x2),x3),...),x_n)
```

which reduces to just `b` in the case that `n`=0.

1. Use `fold` to define the function

   ```
   sum : int list -> int
   ```

   to compute the sum of a list of integers.

2. Use `fold` to define a function

   ```
   reverse : 'a list -> 'a list
   ```

   which reverses the given list.

3. Use `fold` to define the function

   ```
   enqueue_list : 'a queue * 'a list -> unit
   ```

   which enqueues all of the elements of a list in order. (Using the same queue interface as was given to you in Assignment 6.)

3 Pipes Revisited

The producer/consumer design in Assignment 6 has a problem; a the pipe buffer will grow to take up arbitrary amounts of memory if producers produce more quickly than consumers can consume. Implement a version of the pipes code where each pipe has a maximum number of elements it can hold at once; this maximum will be specified as an argument to `newPipe` when the pipe is created. Any producer trying to write to the pipe once it is full should block (stop running) until there is room available in the pipe.

You may want to proceed by modifying the implementation from Assignment 6. Of course, you are permitted to change the definition of the `pipe` type, and you should still use the provided implementation of queues. Place your code in a structure named `Pipe` which satisfies the following signature:

```
signature PIPE =
  sig
     type 'a pipe
     val newPipe   : int -> 'a pipe
     val readPipe  : 'a pipe -> 'a
     val writePipe : 'a pipe * 'a -> unit
  end
```

Your code should be well-commented.