Computer Science 131, Fall 2000
Assignment 5: Static Scope and Modules

Out: Friday, October 20

Due: Friday, October 27, 5:00pm

To complete this assignment, run SML/NJ on turing using `/cs/cs131/bin/sml-cs131-a5`. Put your answers in a file named `assign5.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 3 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-a5`. (This is very like the `sml-cs131-a3` 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 `fun 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, η); it contains two variables, an expression, and an environment.

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

$$
\begin{array}{rcl}
v & ::= & n \\
& | & \overline{\texttt{tt}} \\
& | & \overline{\texttt{ff}} \\
& | & \texttt{Clo}(f, x, e, \eta) \\
\\
e & ::= & v \\
& | & x \\
& | & e_1 + e_2 \\
& | & e_1 < e_2 \\
& | & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
& | & \texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2 \\
& | & e_1\, e_2 \\
& | & \langle e_1, e_2 \rangle \\
& | & e.1 \\
& | & e.2 \\
& | & e_1 :: e_2 \\
& | & \texttt{case } e_1 \texttt{ of nil => } e_2 \texttt{ | } x{::}y \texttt{ => } e_3 \\
& | & \texttt{fun } f(x{:}t_1){:}t_2 \texttt{ is } e
\end{array}
$$

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 3 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.

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

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

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

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.

$$\frac{\begin{array}{c}(\eta, e_1) \Downarrow (\texttt{Clo}(f, x, e_1', \eta_1)) \\ (\eta, e_2) \Downarrow v_2 \\ ((\eta_1, x{=}v_2, f{=}\texttt{Clo}(f, x, e_1', \eta_1)), e_1') \Downarrow v\end{array}}{(\eta, e_1\, e_2) \Downarrow v} \tag{3}$$

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 η_1 (the environment stored in the closure) rather than from η (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 3, doing a search-and-replace to rename the function, making sure that closures evaluate to themselves, and modify the cases for `Fun` and `Apply`. Except for the name change, everywhere, 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 3.

2 Modules

1. Define a signature `STACK` that provides a suitable interface for an implementation of stacks. This should include an (abstract) type `'a stack`, the type of stacks containing values of type `'a`, as well as (at least) operations to push a value onto a stack, pop a value off the stack, a way to get an empty stack, and test whether a stack is empty. Your interface should be suitable for an persistent implementation (where pushing or popping a stack yields a new deeper or shallower stack but does not change the original stack).

2. Implement a structure `Stack` that satisfies the `STACK` signature. The easiest representation for stacks is simply a list.

3. Define a signature `IMPSTACK` that provides an alternative interface suitable for an ephemeral implementation of stacks (where the push and pop operations work not by returning an new stack, but by modifying the given stack).

4. Implement a structure `ImpStack` that satisfies the `IMPSTACK` signature. The easiest representation for such a stack is a reference containing a list. (Be careful about empty stacks.)