Computer Science 131, Spring 2001
Assignment 4: Environments

Out: Wednesday, February 13

Due: Monday, February 18, beginning of class

To complete this assignment, retrieve the files `assign4.sml`. Complete this file and turn it in; as always, your file should contain no syntax or type errors. The submission process is the same as before: (`cs131submit assign4.sml`).

$$
\begin{array}{rcl}
\mathit{value} & ::= & \mathit{num} \\
& | & \mathit{bool} \\
& | & \texttt{(}\mathit{var}\texttt{)=>}\mathit{exp} \\
& | & \langle \mathit{value}_1, \mathit{value}_2 \rangle \\
\\
\mathit{exp} & ::= & \mathit{value} \\
& | & \mathit{var} \\
& | & \mathit{exp}_1\texttt{+}\mathit{exp}_2 \\
& | & \mathit{exp}_1\texttt{==}\mathit{exp}_2 \\
& | & \mathit{exp}_1 \texttt{ ? } \mathit{exp}_2 \texttt{ : } \mathit{exp}_3 \\
& | & \texttt{let } \mathit{var} \texttt{ be } \mathit{exp}_1 \texttt{ in } \mathit{exp}_2 \\
& | & \mathit{exp}_1\ \mathit{exp}_2 \\
& | & \langle \mathit{exp}_1, \mathit{exp}_2 \rangle \\
& | & \texttt{fst } \mathit{exp} \\
& | & \texttt{snd } \mathit{exp}
\end{array}
$$

Figure 1: Abstract Syntax for This Assignment

---

1 Environments (80%)

The interpreter you have seen in class has the advantage of a simple definition, but it's not terribly efficient; it will repeatedly re-traverse large sections of the code each time a substitution occurs.

An implementation can avoid substitution by simply remembering the current values of variables (that is, by maintaining an *environment*, or lookup table, which maps variable names to values). Instead of replacing a variable with a substitution operation, we just record the value of this variable in the environment. When we later come across a variable, we look up its value in the environment.

The `assign4.sml` file defines the type `'a env` (environments associating values of type `'a` with variable names) and three useful definitions:

```
empty  : 'a env
lookup : 'a env * string -> 'a
extend : 'a env * string * 'a -> 'a env
```

The value `empty` is an empty lookup table. The function `lookup` takes an environment and a string and returns the value associated with that string in the environment. (The `lookup` function will raise the `Error` exception when given a string not in the given environment.) Finally, the `extend` operation creates a new environment by adding an association between the given string and the given value to a pre-existing environment. (This operation is "functional" in that a new environment is created, while the environment passed to `extend` is not modified!) Note that extend can cause "shadowing" in environments, if it is given a value for a string already present in the given environment. For example, given

```
val env1 : int env = extend(empty, "a", 1)
val env2 : int env = extend(e1, "b", 2)
val env3 : int env = extend(e2, "a", 3)
```

the expression `lookup(env1,"a")` evaluates to `1`, and the expression `lookup(env2,"a")` evaluates to `1`, but the expression `lookup(env3,"a")` evaluates to `3`. Both `lookup(env2,"b")` and `lookup(env2,"b")` evaluate to `2`, while `lookup(env1,"b")` raises an exception.

This alternate implementation can also be described with a (different) big-step semantics. The relation $(env, exp) \Downarrow value$ inductively defined by the following inference rules can be read as "the expression *exp*, which may have free variables whose values are given by the environment *env*, evaluates to *value*". Most of the rules are very similar to those seen in class:

$$\frac{}{(env, value) \Downarrow value} \tag{1}$$

$$\frac{(env, exp_1) \Downarrow \mathtt{m}_1 \qquad (env, exp_2) \Downarrow \mathtt{m}_2}{(env, exp_1 \texttt{+} exp_2) \Downarrow \mathtt{m}_1 \oplus \mathtt{m}_2} \tag{2}$$

$$\frac{(env, exp_1) \Downarrow \mathtt{m}_1 \qquad (env, exp_2) \Downarrow \mathtt{m}_2}{(env, exp_1 \texttt{==} exp_2) \Downarrow \mathtt{m}_1 \equiv \mathtt{m}_2} \tag{3}$$

$$\frac{(env, exp_1) \Downarrow \texttt{true} \qquad (env, exp_2) \Downarrow value_2}{(env, exp_1 \texttt{ ? } exp_2 \texttt{ : } exp_3) \Downarrow value_2} \tag{4}$$

$$\frac{(env, exp_1) \Downarrow \texttt{false} \qquad (env, exp_3) \Downarrow value_3}{(env, exp_1 \texttt{ ? } exp_2 \texttt{ : } exp_3) \Downarrow value_3} \tag{5}$$

$$\frac{(env, exp_1) \Downarrow value_1 \qquad (env, exp_2) \Downarrow value_2}{(env, \langle exp_1, exp_2 \rangle) \Downarrow \langle value_1, value_2 \rangle} \tag{6}$$

$$\frac{(env, exp) \Downarrow \langle value_1, value_2 \rangle}{(env, \texttt{fst } exp) \Downarrow value_1} \tag{7}$$

$$\frac{(env, exp) \Downarrow \langle value_1, value_2 \rangle}{(env, \texttt{snd } exp) \Downarrow value_2} \tag{8}$$

In the two rules where we had previously specified a substitution, we now merely put the value of the variable in the environment. In the following two rules, the notation $env, var{=}value$ stands for the environment which is exactly like env except that it associates the $value$ with the var. (That is, the result of calling `extend(`env`,`var`,`$value$`)`.)

$$\frac{(env, exp_1) \Downarrow value_1 \qquad ((env, var{=}value_1), exp_2) \Downarrow value_2}{(env, \texttt{let } var \texttt{ be } exp_1 \texttt{ in } exp_2) \Downarrow value_2} \tag{9}$$

$$\frac{(env, exp_1) \Downarrow (\texttt{(}var\texttt{)=>}exp_3) \quad (env, exp_2) \Downarrow value_2 \quad ((env, var{=}value_2), exp_3) \Downarrow value}{(env, exp_1\ exp_2) \Downarrow value} \tag{10}$$

Finally, since we're not substituting variables away any more, it is no longer an error to come across a variable by itself; instead, we look up the value of this variable in the environment. In this rule the notation $env(var)$ stands for the value which the environment env associates with var. (That is, the result of `lookup(`env`,`var`)`.)

$$\frac{}{(env, var) \Downarrow env(var)} \tag{11}$$

Write the function

```
evalEnv : (absyn env) * absyn -> absyn
```

which takes an environment (mapping variable names to values) and an expression, and evaluates that expression as in the inference rules above. This function should also raise the `Error` exception if the expression cannot be evaluated.

To test this function, one can call `evalEnv` with the `empty` environment and the abstract syntax for a program.

The file `assign4.sml` contains the code for several cases of this function. Fix this code so that it returns the correct answer for the remaining cases instead of raising the `Unimplemented` exception.

2 Questions of Scoping (20%)

Although it's not immediately obvious, the semantics defined in the above 11 inference rules is subtly different than the original semantics given in class. (That is, the rules in this assignment define a different language than than the rules given in class!) Thus, on some programs your new interpreter will not give the same results as the original `eval` function.

Test your interpreter on the following program, which you saw in the previous assignment.

```
let x be 1 in
 (let addx be ((y) => y+x)  in
     (let x be 4 in
          (addx x)))
```

An implementation of `evalEnv` that matches the rules given above should yield an answer of 8. Yet according to the semantics that used substitution, the answer was supposed to be 5. In an SML comment, explain how this difference arises.