Computer Science 131, Spring 2002
Assignment 5: More Interpreter Extensions

Out: Wednesday, February 20

Due: Wednesday, February 27, beginning of class

To complete this assignment, you are to modify the environment-based interpreter from Assignment 4. You may use your own, if you believe it to be correct, or you may start from the sample solution available via the course web pages. Put all your answers into a file named `assign5.sml` containing your answer, and then `cs131submit assign5.sml`. (It's actually a good idea to submit early and often, to prevent accidental losses of your work.)

1 Arithmetic (20%)

Extend your interpreter to include the other binary arithmetic operators (subtraction, multiplication, etc.) and the other comparison operators. Obviously this will require modifying both the `absyn` datatype and the interpreter; there are two obvious ways of extending `absyn`:

- Adding a new datatype constructor to the `absyn` datatype for each new operator
- Having one or two cases in the definition of `absyn` that handle arithmetic and comparison operations, as in Assignment 1.

Choose whichever method seems best to you.

2 Recursion (15%)

Another argument for having dynamic scope is that one "automatically" gets recursive function definitions for free; this was a much stronger argument before people figured out several other ways to do recursion in statically-scoped languages. Define the variable `facttest` of type `absyn` to be the SML representation of the following code:

```
let
   fact  be  (n) =>   (n <= 0) ? 1 : n*fact(n-1)
in
   fact 4
```

and verify that your interpreter returns `24`.

3 Imperative Extensions (50%)

Modify your `evalEnv` to add two new sorts of expressions: assignment (*var* `:=` *exp* as in C/C++/Java/etc.) and sequences of two expressions (*exp* `;` *exp* as in SML). Assignment in your interpreter should not only change the current value of the variable, but (being an expression that has to evaluate to something) should also return this same value. Hence, for example,

```
let x be 0 in
   (x := 7) * 2
```

should evaluate to `14`, and more usefully the expression

```
x := (y := (z := 3))
```

should have the effect of setting `x` and `y` and `z` all to be `3`.

As another example, the code

```
let
   x  be  1
in
   ((x := x*2 + 1) ; x)
```

should evaluate to `3`.

The easiest way to do this is to change the environment argument, so that instead of mapping variable names to (abstract syntax representations of) values, it maps variable names to SML references containing such values. Your interpreter will then have type

```
((absyn ref) env) * absyn   ->   absyn
```

This change means that any part of the interpreter which had previously added new variables and their values to the environment must now associate variables with new ref cells; conversely, code that computes the value of a variable by itself must now return the *current* value of that variable, which will be inside a ref cell in the environment. The `absyn` datatype must be extended with two new cases for assignment and sequencing expressions, and you will need to add corresponding cases to the interpreter.

You may assume that variables will not be assigned to unless they are already in the environment (added via `let` or as the parameter of a function).

4 Imperative Loops (15%)

Define the SML variable `facttest2` of type `absyn` to be the SML representation of the abstract syntax for the following program:

```
let  ans  be  1  in
   let fact be (n) => ((n<=0) ? 0 : ((ans := ans*n) ; fact(n-1))) in
      ((fact 4); ans)
```

and verify that this also returns `24`.