CSE 341, Autumn 2005, Assignment 2
ML - Simple Expression Evaluator

Due: Thurs October 13, 10:00pm

Write and test an ML function `eval` that evaluates simple ML integer expressions, involving integer constants, variables, addition (+), subtraction (−), multiplication (`*`), unary negation (`~`), and `let` expressions to bind variables.

If there is an unbound variable in an expression being evaluated, raise an `UnboundVariable` exception.

To simplify the problem, you don't need to parse ML programs. Instead, `eval` takes two arguments: a datatype representing the expression, and an environment in which the expression should be evaluated. Define a new type representing expressions — the datatype `exp` in the code example for Lecture 4 is an excellent starting point for your type, and the function `eval` is a starting point for the `eval` function that you should write. (However, your `eval` function should take two arguments rather than one, namely, the expression to evaluate, and also the environment.)

For example, the ML expression `x+3` could be represented as:

```
Add(Var("x"),Constant(3))
```

Here's a more complex example. The ML expression

```
let val x = 3;
    val y = x+1
in
    x+y
end;
```

would be represented as:

```
Let([Bind("x",Constant(3)), Bind("y",Add(Var("x"),Constant(1)))] ,
    Add(Var("x"),Var("y")))
```

So we're representing the ML `let` expression as the data constructor `Let`, followed by a list of bindings, and then the expression that is in the body of the `let`. Each binding is in turn represented using the data constructor `Bind` followed by a string (the name of the variable), and an expression (which is evaluated to get the value for the new variable).

Test Cases:

Demonstrate your `eval` function on the following examples (at least). These are written here in standard ML — translate them to the representation described above.

```
3
```

```
3+4*10
```

```
let val x = 3;
    val y = x+1
in
    x+y
end;


(* this example should raise an exception *)
let val x = 3;
    val y = x-10
in
    x+y+z
end;


let val x = 3;
    val y = ~x
in
    let val x = 100
    in
        x+y
    end
end;


(* and finally, a really horrible example ... but if your function works on
this one, you've nailed manipulating the environment! *)
let val x = 3;
    val y = x+1
in
    let val x = x+y;
        val y = x+y
    in
        x+y
    end
end;
```

Hints:

Represent the environment as a list of pairs of strings and ints. For example, `[("x",3), ("y",4)]` represents an environment in which the variable `x` is bound to 3, and `y` is bound to 4. To look up the value of a variable, search for the first match in the list.

Solve the problem in stages. First, write and test a function that just handles evaluating expressions involving constants (no variables). This should be easy. Next, write and test a `lookup` function that takes a string (representing the name of a variable) and an environment, and looks up the binding for that variable in the environment and returns the result. For example,

```
lookup("x", [("x",3), ("y",4)])
```

should return 3.

You'll want to define an exception `UnboundVariable` — to do this, just write

```
exception UnboundVariable
```

To raise the exception, use the expression `raise UnboundVariable`.

Then add code to handle variables, but not `let`. (To test this part of the code, invoke `eval` by feeding it an appropriate environment — normally, otherwise, when you invoke `eval` from the command line, the environment will be the empty list.) Finally, add support for `let`.

To make it quicker to test your code, in a file (either your solution or another file) define a set of variables that contain expressions and the results of evaluating them, for example:

```
val exp1 = Let([Bind("x",Constant(3)), Bind("y",Add(Var("x"),Constant(1)))] ,
           Add(Var("x"),Var("y")));
val e1 = eval(exp1,[]);
```

Then at the command prompt you can just type `exp1` or `e1`.

You don't need to write that much code for this assignment — excluding blank lines, comments, and test cases, mine was under 30 lines. However, you may find that figuring out how to handle `let` and environments is tricky. (On the positive side, you'll be working with core computer science concepts, such as binding, scope, and the like — this is excellent material to understand thoroughly.)

You will very likely need to define mutually recursive functions. To do this, use a series of definitions joined by `and`. (See Section 3.2.3 in the text.)

Finally, here's a bit more on environments. As noted above, when you invoke `eval` from the command line, the environment would normally be the empty list. Then, as the expression is evaluated, more complex environments will be constructed as you evaluate `let` expressions. Suppose you've defined `exp1` and `e1` as above.

The expression `eval(exp1,[]);` evaluates the expression in the empty environment — the result should be 7.

Let's trace through how the environment is manipulated. The outer environment is `[]`. As your function processes the bindings in the `let`, it will first construct an environment that binds `x`:

```
("x",3) :: []
```

Then it will evaluate `var y = x+1` in this new environment. Then it (recursively) constructs yet another environment:

```
("y",4) :: [("x",3)]
```

which is used when evaluating `x+y`.

Let expressions can of course be nested, just as in ML. For example, consider:

```
let val x = 3;
    val y = x+1
in
    let val x = 100
    in
        x+y
    end
end;
```

When we're evaluating the expression `x+y`, the environment will be `[("x",100), ("y",4), ("x",3)]`

When we're finding the value of `x` when evaluating `x+y`, we search the list for the first match, namely `("x",100)`, and so we get a value of 100 for `x` and 4 for `y`. This is exactly the behavior we want.

Turnin:

As with HW 1, turn in two files: the source listing for your functions (all in one file), and a log file showing the results of testing the functions.

Extra Credit:

1. In some languages, for example Scheme, `let` has a different semantics than in ML. In Scheme, all of the expressions in the binding part of the `let` are evaluated in the outer scope, and then the variables are bound, rather than evaluating the bindings sequentially. Here's an example. In ML the following expression evaluates to (100,100).

   ```
   let val x = 3
   in
       let val x = 100;
           val y = x
       in
           (x,y)
       end
   end;
   ```

   In the outer scope, you bind `x` to 3. Then in the inner scope, you bind `x` to 100, and then find the binding for `y` using this new binding for `x`. With the Scheme semantics, however, you get (100,3) — when computing the value for y, you use the `x` from the outer scope.

   Modify your `eval` function to use the Scheme semantics rather than the ML semantics. Demonstrate it working on appropriate test cases.

2. Support real numbers as well as integers. In addition to addition, subtraction, and unary negation for reals as well as integers, implement the following functions in your interpreted language: `sqrt`, `real`, and `round`.