CSE 341, Autumn 2005, Assignment 3
ML - MiniML Interpreter

Due: Thurs October 27, 10:00pm

Note: This is a much longer assignment than anything we've seen up until now. You are given two weeks to complete this assignment, but I strongly encourage you to start early. Also, because this is a much more in-depth project, it will be worth three times as much as either of the first two homeworks.

In this assignment, we will be writing an interpreter for MiniML, a subset of the ML language. We will be building on assignment 2, so you are strongly encouraged to use your solution to that as a starting point. (You can also make use of the sample solution to assignment 2 if you would like, perhaps studying it to clean up your own solution before using your code as a basis for this assignment.)

To begin with, in assignment 2 we had to implement a datatype definition looking something like this:

```
datatype exp =
    Var of string
  | Constant of int
  | Add of exp * exp
  | Sub of exp * exp
  | Mult of exp * exp
  | Negate of exp
  | Let of stmt list * exp
and
  stmt = Bind of string * exp;
```

This was a good start, but now we want more functionality! Rather than simply evaluating expressions, we want to eventually get a full-blown interpreter of a subset of ML, which we'll call MiniML. The first step toward this is to improve the `exp` datatype so that it includes not just integers but booleans as well. So instead of the `Constant` type, we want two types – `Int` and `Bool`.

(Note that this means we need to change our environment to store `string * exp` pairs rather than `string * int` pairs. Although theoretically this means that the environment could store complex expressions like `Add(Int 3, Int 4)`, in reality the only `exp` types which should go in the environment are those that cannot be simplified further, such as `Int` and `Bool`. See the hints section for more info on this.)

Now that we have booleans, let's add a few functions that take advantage of them. First, comparison operators `LessThan`, `GreaterThan`, and `EqualTo`. Also, we'll want to do basic boolean logic, so we'll need `And`, `Or`, and `Not`. Another useful ML operation is `IfThenElse`, which will take three arguments and, if the first one evaluates to `Bool(true)`, return the second argument; otherwise it'll return the third argument.

At this point you're probably thinking that `exp` is getting pretty big, but we aren't done yet! Another important feature in ML is list manipulation. We've already seen in the quiz sections that we can build our own list type, and that's exactly what we're going to do here. So, let's add the following types to our `exp` definition: `Nil`, `Cons`, `Hd`, `Tl`, and `Null`. (In this case, `Nil` stands for the ML constant `nil`, or `[]`, while `Null` stands for the ML function `null` which returns true if a list is empty.)

Now that we've got our expanded `exp` datatype, let's extend the `eval` function from assignment 2 so that it handles these additional types. Instead of returning an `int`, make `eval` return an `exp` type in as simplified a form as possible. (e.g., If the result is an int, return `Int(n)` for some `n`, if the result is a bool, return `Bool(true)` or `Bool(false)`; if it's a list, return either `Nil` or a `Cons` object, etc.)

Just like in assignment 2, if there is an unbound variable in the expression being evaluated, raise an `UnboundVariable` exception. If you ever encounter mismatched types, such as trying to add an `Int` and a `Bool`, raise a `TypeError` exception.

For example, the following code should result in `r` evaluating to `Int(4)`.

```
val e = Let([Bind("x", Int 3), Bind("y", Add(Int 5, Var "x"))],
            IfThenElse(LessThan(Var "x", Var "y"),
                       Add(Var "x", Int 1),
                       Sub(Var "y", Int 1)));

val r = eval(e, [("x", Int 12)]);
```

Up until now, the only type of `stmt` object we've seen is `Bind`, which just binds a value to a variable. This is all fine and dandy, but the real `fun` of ML lies in writing your own functions. (Hahaha... I crack me up sometimes.) We want the same *function*ality in MiniML! (Ok, that was just bad.) So, let's extend the `stmt` type to include not only `Bind` but `Fun` as well.

Like a `Bind`, a `Fun` also takes a `string` to identify the variable name to bind the function to. In addition, a `Fun` needs a list of argument names (also represented as `string`s), and it needs an expression, of type `exp`, to evaluate.

The tricky thing about functions, however, is that they are evaluated in the environment in which they were declared, not the one in which they're called. We've already seen this sort of thing in class, but just as a review, `x` will be bound to `11` rather than `16` at the end of this code:

```
val x = 3;
val y = 6;
fun plusx y = x + y;
val x = 8;
val y = 13;
val x = plusx(x);
```

What this means is that, when we encounter a function declaration (via a `Fun` statement), we need to save the environment. This is achieved through a *closure*. We discussed closures in lecture, but to recap: A closure is essentially a function tied to the environment in which it was declared. When a function is invoked, what SML is really doing is extending the environment of its closure to include the function parameters, and then evaluating the function expression in that environment.

Closures are essentially expressions that just happen to be functions. When you declare functions in ML, the interpreter often spits back at you a statement similar to `var f = fn : int -> int`. Think of the `fn` as signifying that the value of `f` is a closure.

Since closures are expressions, what this means is we need to add `Closure` to our `exp` datatype. Remember: closures take a variable name (which will be the name of the function), a list of parameter names, an expression to evaluate, and an environment in which to evaluate them.

This is all fine and dandy, but what good are functions if we can't call them? So now we also want to add one final type to `exp` (I promise, this is it!), called `ApplyFun`. `ApplyFun` takes an `exp` which should evaluate to a closure, and also takes a list of `exp`s which correspond to the parameters to that closure.

When `eval` encounters an `ApplyFun`, the first thing it should do is evaluate the first argument and make sure it's a `Closure`. Then all of the supplied function arguments need to be evaluated and added to the closure's environment. And in order for a function to be able to call itself recursively, it needs to be in its own environment as well. This can be done by adding a binding between the function name (which should be part of the `Closure`) and the `Closure` itself to the environment. Finally, after the closure's environment is extended with the calling arguments and with itself, the closure's expression (the function body) is evaluated in this new environment.

It's important to keep in mind a distinction between expressions and statements in MiniML. An expression evaluates to a value, while a statement generally binds an expression (which could be a function) to a variable and adds this binding to the environment. We've already written `eval`, which handles `exps`. Now we need to write a corresponding function for `stmts`. Thus, we should implement a function `exec` which, when given a list of `stmt` and an environment, returns that environment extended with all the new bindings. For example, suppose we have the following code:

```
val prog = [
  Bind("x", Int 3),
  Bind("y", Int 6),
  Fun("plusx", ["y"], Add(Var "x", Var "y")),
  Bind("x", Int 8),
  Bind("y", Int 13),
  Bind("x", ApplyFun(Var "plusx", [Var "x"]))
  ];

val e = exec(prog, []);
```

The environment returned by `exec` should be equivalent to:

```
val e = [("x",Int 11), ("y",Int 13), ("x",Int 8),
         ("plusx",
             Closure ("plusx", ["y"],
                      Add(Var "x", Var "y"),
                      [("y",Int 6), ("x",Int 3)])),
         ("y",Int 6), ("x",Int 3)];
```

Pretty-Printing

If you've made it this far, you're doing well. You can almost see the light at the end of the tunnel... Almost...

So far, we've got a function `exec` which takes a list of `stmts` and evaluates them in a given environment. Thus, we've essentially written an interpreter for our MiniML language. However, the next thing for us to do is turn MiniML into real ML. In order to do this, we need to write a function called `prettyprint` which, when given a list of `stmts`, actually prints out *valid, executable SML code* for them.

To take our recent example, pretty-printing the following:

```
val prog = [
  Bind("x", Int 3),
  Bind("y", Int 6),
  Fun("plusx", ["y"], Add(Var "x", Var "y")),
  Bind("x", Int 8),
  Bind("y", Int 13),
  Bind("x", ApplyFun(Var "plusx", [Var "x"]))
  ];

prettyprint(prog);
```

Should output something like this:

```
val x = 3;
val y = 6;
fun plusx (y) =
  x + y;
val x = 8;
val y = 13;
val x = plusx x;
```

Note that, unlike with `exec` and `eval`, where there is a correct answer to each function call, the output of `prettyprint` is much more flexible. All it has to do is be equivalent SML code; you're free to control the use of parentheses, spacing, indentation, etc. A perfectly valid (though less pretty) alternative output of the same call to `prettyprint` would be:

```
val x=(3);
val y=(6);
fun plusx(y)=((x)+(y));
val x=(8);
val y=(13);
val x=(plusx(x));
```

As a hint, you can string together multiple statements in a function body by separating them with `;` and enclosing the whole thing in parentheses. For example, consider the following function:

```
fun printpair (x, y) =
   (print "("; print (Int.toString x);
    print ", ";
    print (Int.toString y); print ")\n");
```

If we call `printpair(12,4)`, the output is:

```
(12, 4)
```

The analog to `Int.toString` for booleans is `Bool.toString`.

Test Cases

In addition to the examples we've already seen so far, you should make sure that your `exec` and `prettyprint` functions work on the following test cases (though during grading, we'll certainly test on others as well). Again, the `prettyprint` output can vary, as long as it's equivalent SML code.

```
val prog = [
  Bind("x", Int(4)),
  Bind("y", Let([Bind("z", Add(Var("x"), Int(2)))], Mult(Var("x"), Var("z"))))
];

val e = exec(prog, []);

prettyprint(prog);
```

Should yield the following value for e:

```
val e = [("y",Int 24), ("x",Int 4)];
```

And the following pretty-print output:

```
val x = 4;
val y = let
  val z = x + 2;
in
  x * z
end;
```

Another example, using booleans and a user-defined function:

```
val prog = [
  Bind("l", Cons(Bool false, Cons(Bool false, Cons(Bool false, Nil)))),
  Fun("or_list", ["l"], IfThenElse(Null(Var "l"), Bool false,
      Or(Hd(Var "l"), ApplyFun(Var "or_list", [Tl(Var "l")])))),
  Bind("r", ApplyFun(Var "or_list", [Cons(Bool true, Var "l")]))
  ];

val e = exec(prog, []);

prettyprint(prog);
```

After which e should be

```
val e = [("r",Bool true),
         ("or_list",
            Closure ("or_list", ["l"],
               IfThenElse(Null(Var "l"), Bool false,
                  Or(Hd(Var "l"), ApplyFun(Var "or_list", [Tl(Var "l")]))),
               [("l", Cons(Bool false, Cons(Bool false, Cons(Bool false, Nil))))])),
         ("l", Cons(Bool false, Cons(Bool false, Cons(Bool false, Nil))))];
```

And the pretty-print output should be

```
val l = false :: false :: false :: nil;
fun or_list (l) =
  if null l
  then false
  else (hd l) orelse (or_list (tl l));
val r = or_list (true :: l);
```

And here's a considerably more complex example, showing both number and list manipulation, recursion, and nested functions.

```
val prog = [
  Fun("sum_range", ["a", "b"],
   Let([
        Fun("range", ["a", "b"],
          IfThenElse(LessThan(Var "a", Var "b"),
                     Cons(Var "a", ApplyFun(Var "range", [Add(Var "a", Int 1), Var "b"])),
                     Nil)),
        Fun("sum_list", ["l"],
          IfThenElse(Not(Null(Var "l")),
                     Add(Hd(Var "l"), ApplyFun(Var "sum_list", [Tl(Var "l")])),
                     Int(0)))
       ],
       ApplyFun(Var "sum_list", [ApplyFun(Var "range", [Var "a", Var "b"])]))),
  Bind("r", ApplyFun(Var "sum_range", [Int 3, Int 12]))
  ];

val e = exec(prog, []);

prettyprint(prog);
```

The result is that `e` is bound to

```
val e = [("r",Int 63),
         ("sum_range",
            Closure ("sum_range",["a", "b"], Let #, []))];
```

Where `#` is a large expression containing the entire contents of the `Let` expression.

The pretty-print output of this is

```
fun sum_range (a, b) =
  let
    fun range (a, b) =
      if a < b
      then a :: range (a + 1, b)
      else nil;
    fun sum_list (l) =
      if not (null l)
      then (hd l) + (sum_list (tl l))
      else 0;
  in
    sum_list (range (a, b))
  end;
val r = sum_range (3, 12);
```

Hints

Like in assignment 2, you will do best to solve the problem in stages. First, extend the definition of `exp` to include each of the new expression types. Then extend `eval` so that it can do comparisons (`LessThan`, `GreaterThan`, and `EqualTo`). Then implement boolean logic (`And`, `Or`, and `Not`). Then add list support (`Nil`, `Cons`, `Hd`, `Tl`, and `Null`). Test each of these thoroughly to make sure `eval` is doing what you expect it to do.

Then extend the `stmt` datatype to include `Fun` and the `exp` datatype to include `Closure`. At this point you can implement the `exec` function, which handles both `Bind` and `Fun` declarations.

Once you've done this, add support for `ApplyFun` to `eval`. Don't forget to support recursion!

A bit more on environments: the only types of `exp`s that should show up in your environment are those which cannot be simplified further. In MiniML, these would be `Int`, `Bool`, `Cons`, `Nil`, and `Closure`. Also, in the case of `Cons`, make sure that the head and tail of the list are in fully simplified form.

Wait until you've finished with `eval` and `exec` before starting on `prettyprint`. The most important part of pretty-printing is a function which pretty-prints an `exp`. You'll want to have the actual `prettyprint` function call this auxiliary function on the `exp` associated with each `Bind` and `Fun`.

Start off by not worrying about extraneous parentheses or indenting. Just get output that works. However, keep in mind that to get full credit your code does need to look like someone could've plausibly written it by hand. Remember, the output must be valid SML code which, when run in an SML interpreter, does the same thing as calling `exec` on the original code.

This is a much more in-depth project than assignment 2, not the sort of thing you can hack together at the last minute. My solution to this has about 150 lines of code. Start early and it'll be a much more pleasant and rewarding process.

Turnin:

As with HW 1 and HW 2, turn in two files: the source listing for your datatypes and functions (all in one file), and a log file showing the results of testing the functions.

Extra Credit:

Our implementation of MiniML leaves out many useful features of the ML language. For extra credit, try doing some of the following:

1. In the assignment, the type of the environment is `(string * exp) list`. This is a mild abuse of the `exp` type, since in reality the environment should only hold `Int`, `Bool`, `Cons`, `Nil` and `Closure`. Clean this up by defining another datatype, `miniML_data`, that holds exactly this information, and modify your solution so that the environment is of type `(string * miniML_data) list`. (You will need to make other changes as well to accomodate this.) [This extra credit item was put in just to humor the instructor ...]
2. Add anonymous functions. This should be a straightforward extension of supporting named functions.
3. Add tuples and records.
4. Support pattern-matching, either in function headers or using `case`.
5. Add static type checking. Define a function `typecheck_expression` that accepts a MiniML expression and return its type, or `BadType` if it's ill-typed. For example, `IfThenElse(Bool true, Int 3, Nil)` should have type `BadType`. Then define another function `typecheck_program` that type-checks a program (a list of statements) and returns `true` if none of the expressions are of type `BadType`.