

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 *functionality* 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 `strings`), 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 `exps` 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 `exprs`. 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"]))  
];
```