EECS 345: Programming Language Concepts

Interpreter Project, Part 3

Due Wednesday, March 30

In this homework, you will expand on the interpreter of part 2 adding function definitions. We still assume all variables store integers and boolean. Likewise, all functions will only return integers and boolean.

While normal C does not allow nested functions, the gcc compiler *does* allow nested functions as an extension to C, so let's implement them!

For those seeking a small extra challenge: try implementing both the *call-by-reference* and the *call-by-value* parameter passing styles.

An example program that computes the greatest common divisor of two numbers is as follows:

```
var x = 14;
var y = 3 * x - 7;
function gcd(a,b) {
  if (a < b) {
    var temp = a;
    a = b;
    b = temp;
  }
  var r = a % b;
  while (r != 0) {
    a = b;
    b = r;
    r = a % b;
  }
  return b;
}
function main () {
  return gcd(x,y);
}
```

Here is another example program that uses recursion:

```
function factorial (x) {
  if (x == 0)
    return 1;
  else
    return x * factorial(x - 1);
}

function main () {
  return factorial(6);
}
```

Note that only assignment statements are allowed outside of functions. Functions do not have to return a value. The parser will have the following additional constructs:

```
function a(x, y) {          =>   (function a (x y) ((return (+ x y)))
  return x + y;
}

function main () {          =>   (function main () ((var x 10) (var y 15) (return (funcall gcd x y))))
  var x = 10;
  var y = 15;
  return gcd(x, y);
}
```

The final value returned by your interpreter should be whatever is returned by `main`.

Nested functions can appear anywhere in the body of a function. Any name in scope in a function body will be in scope in a function defined inside that body.

```
function main() {
  var result;
  var base;
```

```
  function getpow(a) {
     var x;

     function setanswer(n) {
        result = n;
     }

     function recurse(m) {
       if (m > 0) {
         x = x * base;
         recurse(m-1);
       }
       else
         setanswer(x);
     }

     x = 1;
     recurse(a);
  }
  base = 2;
  getpow(6);
  return result;
}
```

We will use a similar style as C++ for call-by-reference:

```
function swap(&x, &y) {     =>  (function swap (& x & y) ((var temp x) (= x y) (= y temp)))
  var temp = x;
  x = y;
  y = temp;
}
```

Function calls may appear on the right hand side of global variable declaration/initialization statements, but the function (and any functions that function calls) must be defined before the variable declaration. Otherwise, functions that are used inside other functions do not need to be defined before they are used.

It is an error to use call-by-reference on anything other than a variable. For example, if the program contains `swap(x, x + 10)` with the above definition of `swap`, you should give an error because `x + 10` is not a variable.

You do not have to stick to strict functional programming style, but you should avoid global variables because they will make your life harder. A new parser is provided for you, `functionParser.scm`, that will parse code containing functions/methods as in the above examples. To use the parser, type the code into a file, and call `(parser "`*`filename`*`")` as before. To call the parsers from your interpreter code, place the command `(load "`*`parsename`*`")` in the Scheme file.

What your code should do

You should write a function called `interpret` that takes a filename, calls `parser` with the filename, evaluates the parse tree returned by `parser`, and returns the proper value returned by `main`. You are to maintain an environment/state for the variables and return an error message if the program attempts to use a variable before it is declared, attempts to use a variable before it is initialized, or attempts to use a function that has not been defined.

Some hints

Terminology In this interpreter, we will be talking about *environments* instead of *states*. The state consists of all the active bindings of your program. The environment is all the active bindings that are in scope.

1. Note that the base layer of your state will now be the global variables and functions. You should create an outer "layer" of your interpreter that just does M_state functions for variable declarations and function definitions. The declarations and assignments should be similar to what you did in your part 2 interpreter. The functon definitions will need to bind the function closure to the function name where the closure consists of the formal parameter list, the function body, and a function that creates the function environment from the current environment.

2. Once the "outer" layer of your interpreter completes, your interpreter should then look up the `main` function in the state and call that function. (See the next step for how to call a function.

3. You need to create a M_value function to call a function. This function should do the following: (a) create a function environment using the closure function on the current environment, (b) evaluate each actual parameter in the current environment and bind it to the formal parameter in the function environment, (c) interpret the body of the function with the

function environment. Note that interpreting the body of the function should be, with one change, *exactly* what you submitted for *Interpreter, Part 2*. Also note that if you are using boxes, you should not have to do anything special to deal with global variable side effects. If you are not using boxes, you will need to get the final environment from evaluating the function body and copy back the new values of the global variables to the current environment/state.

4. Change the M_state and M_value functions for statements and expressions, respectively, to expect function calls.

5. Test the interpeter on functions without global variables, and then test your functions using global variables. One tricky part with the functions is that, unlike the other language constructs we have created, function calls can be a statement (where the return value is ignored), and an expression (where the return value is used). You need to make sure both ways of calling a function works.

6. Since exceptions can happen anywhere that a function call can occur, you may discover more places that need the throw continuation. If you used call/cc for throw, then you should not have to modify anything else in your interpreter from part 2. If you used tail recursion for throw, you will need to make the M_value functions tail recursive for throw to work correctly.