Assignment 4: The Code Generator

This is due on Friday, May 6, the last day of classes.

Write a code generator for BPL, then relax and enjoy the fact that you have implemented a programming language. Your final program should have some way to specify a BPL file and produce a file of code I can assemble with gcc. I would prefer this to run so that if I give your program file foo.bpl it generates file foo.s.

There is a lot to do for this assignment. Here is a suggested game plan.

A. Write a few assembly language programs that use our runtime environment, to ensure that you know what sort of code you should be generating.

   Translate this into assembler:

```
void main(void) {
        write ( “Hi, Bob!” );
}
```

   Here is another example:

```
int f( int n ) {
        if (n == 0)
                return    1;
        else
                return n * f(n-1);
}

void main(void ) {
        write (f(5));
}
```

B. Write functions that walk through your parse trees to compute the indexes or offsets for parameters and local variables.

C. Write helper functions to format assembly language commands:
   genRegReg( )
   genRegIndirect( )
   genIndirectReg( )
   etc.

D. Write a function that creates the header for your code file. Ultimately this will allocate code for global variables and arrays, but it is enough to start with the .ro section for the I/O string declarations.

Your code generator consists primarily of 3 mutually-recursive functions:

genCodeStatement(TreeNode t) which generates code for this statement

genCodeExpression(TreeNode t) which generates code that evaluates the expression and leaves its value in the accumulator

genCodeFunction(TreeNode t) which generates code for a function declaration.

E. Start with genCodeFunction(TreeNode t). This prints the function name as a label, sets up the stack frame, and then does code for the body of the function. At the end it deallocates the stack frame and returns.

F. Start genCodeStatement(TreeNode t) with the write(x) statement.

G. Start genCodeExpression(TreeNode t) with literal numbers (I call these IntNode trees).

You should now be able to generate working code for

```
void main(void) {
        write(34);
}
```

H. Add arithmetic operators. You should now be able to write (1+1).

I. Add comparison operators.

J. Add if-statements

K. Add function calls. You should now be able to handle programs like

```
int f(int x) {
        return 2+3;
}
void main(void) {
        write( f(34) );
}
```

L. Add variables. Think carefully and draw some pictures before you start the code for this.

Among the many tests you should run at this point are functions that make use of their arguments. If you have implemented functions and variables correctly recursion (such as the factorial function) should work.

M. Add while loops.

N. Add the rest.

Some documents you may want to refer to are:

1. The BPL reference manual
2. Notes on the x86-64 assembly language
3. A Runtime Environment for BPL
4. Using gdb with assembly language
5. Class notes for March 30, April 1, April 4