Code Generation Variables

At the top of your assembly language program you will have a data segment that has directives allocating each global variable and array. For example,

```
.comm   X, 8, 32    allocates an 8-byte integer X
.comm  A, 80, 32  allocates an array of 10 8-byte integers
```

These directives make labels out of the symbols X and A; you can use these labels in direct addressing mode.

When you are generating code you need to be able to find variables and arrays. They could be

- Global
- Parameters (arguments to function calls)
- Local variables (declared in compound statements)

Global variables are allocated in the data segment; parameters and local variables are on the stack. You need to be able to find these things; some pre-processing helps.

I give every variable and array declaration attributes *depth* and *position*.

Global variables all have depth 0; we never use their position.

Parameters all have depth 1; their position is their index in their function's list of parameters (the first one has index 0, the second parameter has index 1, etc.)

Local variables all have depth 2 or more. Their position is a sequential count of the order in which they are declared, incremented only for variables that are active at the same time

For example, consider the following silly function:

```
void f(int n) {
        int  x;
        int y;
        x = 1;
        {
                int z;
                z = 2;
        }
        {
                int w;
                w = x;
        }
```

Here are the depths and positions of the local variables:

```
void f(int n) {
        int  x;   //depth=2, position=0
        int y;    //depth=2, position=1
        x = 1;
        {
                int z;  // depth=3, position=2
                z = 2;
        }
        {
                int w;  //depth=3, position=2
                w = x;
        }
```

Variables z and w can be stored at the same position because they aren't live at the same time.

Note that we will treat an Array of size n as a declaration of n variables, so the following declarations will have the given positions:

```
{
        int x;     // level=2, position =0;
        int A[10];   //level = 2, position = 10
        int y;    // level = 2, position =11;
}
```

We will say more about arrays at some point in the future.

These attributes are created with two functions:

A. findDepthDeclaration(TreeNode t, int level, int count) walks along a list of declarations. For each variable it assigns the current value of count to the position attribute and recurses on t.next with count+1. For the global declarations level is 0; for parameter lists it is 1; for declaration lists of compound statements level is 2 or more.
B. findDepthStatement(TreeNode t, int level, int count) walks along a list of statements. The recursive call to the next element in the list, t.next, has the same arguments level and count. Nothing is done for statements unless they are compound statements. For these findDepthDeclaration( ) is called on the declaration list with level+1 and count. It then recurses into the body of the compound statement with count incremented by the size of the declaration list.

Each function declaration needs an attribute that contains the number of its maximum simultaneous local declarations. This is one more than the maximum position of all of the local variables declared within the body of the function. When the function is called we decrement SP by 8 times this size, to allocate at once all of the local variables.

Now, to access the value stored in variable X:

a) If X has depth 0 (ie. it is global), use symbol X as a label. For example, to puts the value of X into %rax do
   movq X, %rax
b) If X is a parameter (depth 1) its offset from fp is 16+8*X.position.
c) If X is a local variable its offset from fp is -8-8*X.position

Assignments.

Here is some handy terminology. The L-value of a variable is the address where it is stored. The R-value of the variable is the data stored in it.

Here is a sequence of code that handles the assignment foo=a:

a) Generate code to put the L-value of foo into ac.
b) Push ac onto the stack.
c) Generate code to put the value of a into ac.
d) Pop the stack into a temporary register T
e) Move the data in ac into 0(T)

Note that we will treat an Array of size n as a declaration of n variables, so the following declarations will have the given positions:

```
{
        int x;     // level=2, position =0;
        int A[10];   //level = 2, position = 10
        int y;    // level = 2, position =11;
}
```

We will say more about arrays at some point in the future.