As a first step toward constructing a complete C° compiler, I would like you to implement the semantic processing required for C° programs. As you know, semantic processing occurs after syntactic processing. To enable you to work on semantic processing before syntactic processing, I will provide you with object code for a syntactic analyzer for C° that produces syntax trees in the form described by the accompanying An Intermediate Form for C° Programs handout. The details of using this syntactic analyzer are discussed in yet another handout: Working with the C° Parser.

**Required Processing**

As we have discussed in class, during semantic processing:

1. declaration descriptors are created,
2. references to identifier descriptors that had been used to represent identifiers in the output of the syntactic analysis phase are replaced by references to the appropriate declaration descriptors,
3. information about each variable’s type, size and location in memory is stored in its declaration descriptor, and
4. checking for semantic errors is performed.

In addition, in your compiler we will include a bit of “high-level code generation” in semantic processing. Namely, you will replace the subtrees the parser produces to represent variable references with subtrees that more explicitly describe the address arithmetic and memory references required to access variables.

Your code will first have to traverse the type definitions found in the main program creating declaration descriptors for all the types defined there. For each type you will need to create an appropriate declaration descriptor and fill in values for its components. Forward references are not allowed in the type definition section of a C° program so this processing can be performed in a single pass.

For each array type you will need to create both a declaration descriptor for the type itself and a descriptor for an imaginary variable of the array’s element type. You will need to store a pointer to the imaginary variable descriptor in the array’s declaration descriptor. You should also set the `typesize` and `size` fields of the array’s declaration descriptor.

For structure types you will need to create a declaration descriptor for the structure type itself and for each component name associated with the structure. The component descriptors will need both to be stored in a linked list pointed to by the structure type descriptor and entered in the hash table used to locate the appropriate declaration descriptor when processing a reference to a component name. As you process component declarations, you should keep track of the number of memory units required for the structure type. Each component should be assigned a displacement within the structure equal to the sum of the sizes of all previously processed components. You may assume an integer takes a single unit of storage (but you should still use a symbolic name for this quantity).

While processing Nident nodes in this section of the program you should make each Nident node’s `decl` component point to the appropriate declaration descriptor.
While processing these declarations, you should do basic error checking. In particular, you should verify that:

- no name is defined more than once, and
- each name used as an array’s element type or as the type of a structure component is defined earlier in the type section.

When errors are detected, be sure to take actions appropriate to let you later avoid printing spurious error message or, worse yet, crashing. For example, if an array’s element type is undeclared, set the element type in the array’s descriptor to a known value (NULL?) rather than leaving it uninitialized.

Your code should print error messages to the standard error output file if errors are detected here or at any other point in semantic processing. Your error messages should be as informative as possible. Each error message should include the number of the line in the source file on which the error occurred. If appropriate, information such as the name of the identifier involved should be included. In addition to printing a message for each error, you should increment the global variable `errorcount` each time such an error is printed. A declaration for this variable is included in `syntree.h`. The value of this variable will be used later to decide whether or not to proceed with code generation.

Next you will need to process the body of the program. Processing the main program’s body should be identical to processing the body of a function. First you traverse the list of variable declarations. A declaration descriptor must be created for each variable and a pointer to the declaration descriptor for the variable’s type must be stored in the descriptor created. Displacements should be assigned to variables as described in class. Multiple declarations of a single name in a given nesting level or uses of undefined names should again be detected and reported.

Each variable descriptor created must be added to the stack attached to the variable name’s identifier descriptor and added to the open scope’s list of declarations.

The processing of the list of function declarations in a body is a little more complex. Forward references are allowed among functions. So you must make two passes over the function definitions. In the first pass, you will create a declaration descriptor for each variable and a pointer to the declaration descriptor for the variable’s type must be stored in the descriptor created. Displacements should be assigned to the formals. The declaration descriptors for a function’s formals should be collected into a linked list rooted in the function’s declaration descriptor at this point. They should not, however, be placed in the list of declarations associated with the current open scope.

In the second pass, you will examine the bodies of the functions. For each function you will first push a new open scope onto the stack of open scopes and add the function’s formal parameters to the set of declarations processed in the scope. Then, you can process the body of the function (Note: this is where the specification of how to process a body gets recursive).

After processing all the functions in a body, you must process the statement list. This is simply a process of walking about the tree that represents the statement list looking for references to identifiers and replacing each pointer to an identifier descriptor by the appropriate declaration descriptor.

Well, not quite. Two things make this step a bit more complicated. First, you have to do a considerable amount of error checking. Second, you have to handle references to variables correctly.

To guide you in error checking, the following hopefully (but not necessarily) complete list of errors to consider is provided:

- All names used as simple variables in the program are declared in some scope surrounding the use.
- The type of any variable that is subscripted is an array type.
The type of any variable from which a component is selected is a structure type including an appropriate component name.

The type of each expression used as an operand of any of C’s operators is integer.

The type of any expression used as the right hand side of an assignment statement is integer.

The type of the variable used as the left hand side of an assignment statement is integer.

The type of any expression used as the condition in an if or while statement is integer.

The type of any expression used in a return statement is integer.

The name used to identify a function in each function call statement is declared as a function.

The name used to identify a function in each function call expression is declared as a function returning an int.

The number of actual parameters passed to any function equals the number of formals included in the function’s definition.

The type of each actual parameter expression used in a call matches the type of the corresponding formal.

Any actual parameter passed where a call-by-reference parameter is expected is actually an assignable object (i.e. an Nrefvar).

Processing of variable references is complicated in two ways. First, as discussed in class, to correctly resolve a reference to a structure component name you need both the type of the structure variable from which the component is being selected and a pointer to the component name’s identifier descriptor. If the selection is something simple like

\[ r.c \]

this seems easy. You just fetch the descriptor pointed to by the vartype component of \( r \)’s descriptor. Unfortunately, component selections can be more complex. For example, it is legal to say

\[ r.a[f(x)].b.c[m].e.f.g \]

To process the component name \( g \) you need access to the declaration descriptor for the type of the sub-variable \( r.a[f(x)].b.c[m].e.f \).

The way to handle such complex sub-variables is to write a recursive variable processing routine that does any identifier resolution needed within the sub-variable and then returns the declaration descriptor of the sub-variable. If this routine were called to process \( r.a[f(x)].b.c[m].e.f.g \), it would then call itself recursively on \( r.a[f(x)].b.c[m].e.f \). The recursive call would provide the descriptor for the “f” component referenced by \( r.a[f(x)].b.c[m].e.f \) as its return value. The original call could then use this descriptor to access the type of “f” (which better be a structure type). The type of the structure together with the identifier descriptor for \( g \) would make it possible to look \( g \) up in the structure component hash table. Once \( g \) is correctly resolved, the declaration descriptor for \( g \) will be known and can be returned to whomever called the routine to process \( r.a[f(x)].b.c[m].e.f.g \).

The second complication involving variables is that I want you to restructure subtrees that reference variables while you are processing them in the semantic analysis phase. After you are done with this transformation, the subtrees should describe the address arithmetic required to access each variable.
For example, a simple variable like “x” will be represented by an Nrefvar node with an Nident node for “x” as its single child. I would like you to replace this Nident node with an Nplus node with two children. One child should be an Ndisplay node referencing the address of the activation record for the function in which “x” was declared. The other should be an Nconst whose value is equal to the displacement assigned to “x”.

Similar transformations can be performed for Nselect and Nsubs nodes. In fact, both of these will also be replaced by Nplus nodes. In each case, one child of the Nplus will be a subtree for the sub-variable appearing as child[0] of the original node. In the case of an Nselect the other child should be a constant node whose value equals the components displacement. In the case of an Nsubs, the second child will be an Ntimes node multiplying together the subscript expression and a constant equal to the size of the array’s element type.

What makes this all nasty, is that if you try to do both of these in one pass (which you can do), you will be writing a recursive function that wants to return two values:

- the declaration descriptor for the variable being referenced, and
- a pointer to the tree which should replace the original variable reference sub-tree.

This is difficult to do in C because C functions can only return a single value and C does not explicitly support call-by-reference (var) parameters.

You have two options:

- make two passes. First resolve all Nident nodes and then translate the trees on a second pass, or
- fake a var parameter by passing a pointer to the variable you really want to pass.

I think the first option might be cleaner (although I took the second in my implementation).

**Sub-phases**

This phase will be further divided into two sub-parts. First, I would like you to write the code needed to do everything you can without traversing statement list. Then, in the second sub-phase we will traverse and process statement lists. This means that during the first phase you will not have to do much type checking and you won’t have to worry about processing and translating variable sub-trees.