Due Tuesday, February 20.

The next component of your compiler to implement is the semantic analyzer. At this point, its function will be the following:

- Ensure the type correctness of the source program and to incorporate type information into the abstract syntax tree.
- Incorporate scope information into the abstract syntax tree and to differentiate between occurrences of the same identifier in the program that denote references to different objects.
- Catch errors in the source program due to references to undeclared names, and multiple declarations of the same name in the same scope.

You may choose to implement the semantic analyzer as a separate phase that traverses the abstract syntax tree (AST), or as part of your parser by adding additional semantic actions to your existing Bison/JavaCC specification.

The routines provided by the semantic analyzer will be used to associate additional attributes with various nodes of the AST. In order to do this, a symbol table should be created (if you have not already done so) and the data structures used for the AST must be extended to include these attributes.

At this point, the following attributes must be associated with each name in the program:

- The type of the identifier.
- The scope of the declaration of the identifier. Since the source language does not have nested procedures, there are only two levels of scoping – global and local. All procedure and type names are global and variable names can be either global or local.

The type information can be represented by tree representations of type expressions (such as those already in the abstract syntax tree) or the more concise representations (value numbering, etc.) described in the text. Note that nodes representing function and procedure names must also have type information associated with them (i.e. types of parameters, return types).

Symbol Table

The semantic analyzer should create a symbol table so that different attributes can be associated with different declarations of variables with the same name. One possible representation of symbol table entries might be:
In this figure, there are three entries in the symbol table, for the identifiers y, x, and foo. y is a local variable (in the procedure that is currently being type-checked) of type foo. x is both a local variable of type integer in the current procedure and a global variable. The type of the global variable x is an array, so the attribute representing this x might point to the subtree representing that array type in the AST. The identifier foo is the name of a record type, also described by a subtree in the AST.

When the semantic analyzer encounters a declaration (type, variable, procedure, or function), it should install the appropriate new attribute node in the symbol table. When the parser leaves the block in which the declaration occurred, the attribute node should be removed (it is easy for the parser to determine which attribute nodes to remove - just the ones for the new names declared in the block).

Another possible representation for the symbol table could be as described in the Dragon book, where each procedure has its own symbol table.

The Abstract Syntax Tree

The attribute information ("attribute nodes") incorporated into the symbol table should also be incorporated in the abstract syntax tree (AST). In particular, all occurrences of a name (variable references, references to type names, etc.) in the AST should share an attribute node. The attribute node for an identifier will have been created when the identifier was declared, and will have been installed in the symbol table. Thus, any reference to that identifier can point to the attributed node. When the scope of the identifier is exited, the attribute node is deleted from the symbol table, but should remain in the AST. For example, the AST representation for the program fragment:
var x: integer;
begin
  x := x + 1;
end;

might be:

These drawings are just schematic, of course. You are free to use whatever representation you see fit. Just be sure that attribute nodes are shared appropriately. In subsequent phases (e.g. during code generation) these attribute nodes will need to contain other necessary information (locations, offsets, sizes, register info, etc.).

**The Type System**

The source language contains three predefined types, integer, string, and boolean, and two predefined type constructors, array and record. Type equivalence is defined as follows:

- Two types named T1 and T2 are equivalent if T1 = T2 (i.e. they are the same type name) or if T1 was defined as follows
  
  ```
  type T1 = T3;
  ```

  where T2 and T3 are equivalent, or if T2 was defined as follows

  ```
  type T2 = T3;
  ```

  where T1 and T3 are equivalent.

- Two anonymous types are equivalent if they are constructed from the same type constructors and their components are equivalent. For records, this means that the names and order of the fields must be identical and their corresponding types must be equivalent. For arrays, the bounds must be identical and the element types must be equivalent.

Type compatibility is defined in the usual way:
The types of the corresponding formal parameters and actual parameters for procedure and function calls must be equivalent.

The types of the LHS and RHS of an assignment statement must be equivalent. Notice that there is a special case in which the name of a function can be assigned to (in which case the result type of the function and the type of the RHS must be equivalent).

The predefined operators (+, *, etc.) and literals (123, “hello”) carry their usual types.

Notice that some new entries must be entered in the symbol table before compilation begins, namely the entries for integer, string, boolean, true, false, and other predefined identifiers.

**Implementation of Type Equivalence**

This kind of equivalence is easy to implement. If, in your symbol table, the symbol A simply points to a type descriptor (attribute node), then if your type checker encounters:

```
type B = A;
```

it can install B into the symbol table, pointing to the same type descriptor as A.

**The Namespace**

There is only a single namespace where type, variable, and procedure names reside. This means that within a given scope (either global or within a procedure), there can be at most one declaration of each name. For example, one could not declare a type named `bar` and a procedure named `bar` in the same program.