Hints for Assignment 3

What Should the Visitor(s) Do?

Type-checking and semantic-checking the AST requires at least two traversals and hence at least two more visitors to be programmed. Let's call the `TCVisitor1` and `TCVisitor2`.

TCVisitor1:

This visitor more-or-less repeats the same traversal as should be performed by the top-level visitor `TLVisitor`, produced for Assignment 2. That visitor filled in a top-level symbol table with incomplete entries describing which classes existed in the program and what their members were. The entries were incomplete because information about the datatypes of consts, fields and methods was totally omitted.

The job of `TCVisitor1` is to fill in all that missing type information. So it traverses the tree, and when it hits nodes with these tags, it performs the actions sketeched out alongside.

`Program`	visit just the class declarations
`Class`	look up the `CbType` type description for this class in the current namespace; visit each member of the class passing the `CbType` value as a parameter
`Const`	look up the `CbConst` type description for this const in the current class; visit left subtree to discover what datatype it represents; store the type of the left subtree in the `CbConst` description for this const.
`Field`	similar to the above except that the type obtained from the left subtree must be stored into the `CbField` description for each identifier in the list of field names,
`Method`	look up the `CbMethod` type description for this method in the current class; visit child 0 if it's not null to discover the return type of this method; store that return type in the `CbMethod` instance; initialize the list of argument types in the `CbMethod` instance to an empty list; now visit each formal parameter – for each one, determine its type and add that type to the list of argument types for the method.
`IntType`	store `CbType.Int` in the `Type` property of the current node

`CharType`
`StringType`
`VoidType`
	similarly to the above

`Ident`	we are visiting an identifier node in a position where a datatype is required – therefore the identifier has to be the name of a class (anything else is an error so make sure that the identifier doesn't refer to anything wrong such as a const member of the class); look up the class in the current namespace; if found, the `CbClass` instance describing this class type is stored in the `Type` property of the current node.

`Array`	visit the child to obtain the element type of the array; use the `CbType.Array` method to create an array type description and store that in the `Type` property of the current node.

TCVisitor1 does not visit the initialization values in const definitions and it does not visit the bodies of methods. Why not? It's because those parts of the AST may contain uses of identifiers which are the names of class members (both this class and other classes). Since we haven't yet finished filling in the type information for class members, we cannot perform any type checking. That has to be left to the second visitor.

TCVisitor2:

This visitor does visit the righthand sides of const definitions and it does traverse the bodies of methods. The `SymTab` class (provided in the assignment materials) should be useful for typechecking a method body.

To typecheck a method, one should start with a `SymTab` instance which has been cleared to empty. Then the formal parameter names and their types and entered. Now the traversal proceeds through the statement body. Every time a statement list is encountered, a new scope is started before visiting the statements within that list, and the scope is exited afterwards.

All name lookups are performed first in the `SymTab` instance, then amongst the members of the current class, and finally among the names of classes in the program.

Details of how to perform the traversal are left as part of the assignment.

What About Other Semantic Checks?

Verifying that the type used in a cast expression has the correct syntactic structure can be incorporated into `TCVisitor2`.

If there are any other semantic checks to be performed (and which are not implied by type checking), they can also be incorporated into `TCVisitor2`.

What Output Should the Cb Compiler Produce After Assignment 3?

After `TCVisitor1` has traversed the AST, all type descriptions needed by the program's top level should be complete. Therefore a run of the `cbc` program with the `-ns` option (to dump the namespace contents) should produce some detailed output. For the class `Foo` declared as on the left side of Figure 1, the `-ns` option should produce output looki like that shown on the right side.

After the `TCVisitor1` traversal, *some* of the AST nodes have had datatype information stored as their `Type` property values.

After the `TCVisitor2` traversal, *all* AST nodes which represent types or values should have their `Type` properties set to hold datatype information.

Running the `cbc` program with the `-tc` option should cause the AST to be printed after all type checking has been completed. That printout will include the `Type` values associated with nodes whenever the `Type` property does not have a value of `null`.

Most checking that `TCVisitor2` works properly will involve feeding it Cb code which contains semantic errors (type mismatches most of the time) and verifying that a reasonable error message is generated for each distinct error.

Figure 1: CbCode on left, NameSpace display on right

```
class Foo {
  public const int theAnswer = 42;
  public const string hiThere = "hello";
  public const char itsAnX = 'x';
  public int a;
  public string b;
  public char c;

  public static void Main() {
    Foo f;
    f = new Bar();
    int r;
    r = f.Umm(3,4);
  }

  public virtual int Ummm( int a, int b ) {
    System.Console.WriteLine(
      "This is Foo");
    return a+b;
  }
}
```

```
class Foo : Object {
    theAnswer:int
    hiThere:class String
    itsAnX:char
    a:int
    b:class String
    c:char
    static void Main()
    int Ummm(int,int)
}
```