CSC435/535: Assignment 3 (Due: 11:55pm, Thursday 3 July 2014)

Overview

The next stage of the course project is to complete the type checking and semantic analysis phase of the Cb compiler.

Assignment Description

1. Code which implements Assignment 2 is provided. You can choose to use this code instead of your own implementation of Assignment 2, or you can mix and match as you wish.
2. You can implement the type checking and semantic analysis phase using as many visitor classes and as many passes over the AST as you like – as long as your approach is reasonable.
3. After your visitor classes have finished their traversals, the following things must hold:
   - Any datatype description which is in use (i.e. an instance of the `CbType` class or any of its subclasses) must be complete – all the details have been filled in.
     Implied by the above is that the Parent property of every `CbClass` instance in use must be provided. If the Cb program did not specify a parent class, then the parent class is by default `System.Object` (which has been predefined in the `System` namespace). Only the class System.Object can have `null` as the value of its Parent property.
   - Any AST node which represents a value (or can hold a value) must have its `Type` property set to an appropriate value. A node which does not represent a value (such as a statement) should have its `Type` property set to `null`.
   - If a string constant is followed by a dot and an identifier, that identifier has to be the property name `Length`.
   - Every reasonable static check on the type correctness of the Cb program should have been performed. A list of things to check is provided below. Every semantic error which has been found should have generated an error message. Semantic checking should attempt to catch as many errors in the program as reasonably possible, while avoiding a cascade of spurious messages.

Some Type Checking / Semantic Checking Requirements

- There must be no circularities in the class hierarchy – i.e., we cannot have class A listing B as its parent, and class B listing A as its parent.
  Just checking that every parent-to-parent chain ends with class `System.Object` is sufficient.
- The form of the type used in a cast expression must be checked. To avoid problems with the grammar, nonsense is syntactically permitted where a datatype is required. You must check that the supplied type is *not* nonsense, and generate an error message when needed.
- Every operator must be applied to operands with types permitted by the operator's signature. Coercion of types from `char` to `int` (widening) and from a class to an ancestor class are permitted. If in doubt as to what is permitted, try an experiment with C# and see what the C# compiler will accept.

- Passing parameters to methods must be handled in a manner similar to an assignment from the actual parameter to the formal parameter.
- The program must contain at least one method whose name is `Main`, has no arguments, and is marked as `static`. It is not an error to have more than one `Main` method, though it may be worth generating a warning message. Our `Main` method does not take arguments just to simplify the run-time requirements.

Some Type Checking Rules

This is not an exhaustive coverage of what and how to check. It's just some things about C# (and Java too) which you need to be aware of.

- Assignment of an array value (the source) to an array variable (the destination) is valid if both element types are classes, and the source class is the same class as, or a subclass of, the destination class. (This is known as *array covariance*.) However, you cannot assign a `char` array to an `int` array, similarly you cannot assign either a `char` array or an `int` array to an array of `object`. On the other hand, you can assign an array of *absolutely anything* to a variable whose type is `object`.
- The `null` value can be assigned to any variable with a class type or an array type.
- The keyword `this` has the same type as the class in which it is being used.

Some Special Cases for Type Checking

- The Cb language does not allow overloading of methods in the Cb program being compiled. However, we do want to allow access to a few methods in the `System` namespace which are overloaded. These have to be handled as special cases. They are as follows.
  - `System.String.Substring`: Cb allows use of both the forms `s.Substring(strt)` and `s.Substring(strt,len)`. Only the two argument form is entered into the `System` namespace by the supplied initialization code. If you encounter the one argument form, it should be accepted.
  - `System.Console.WriteLine`: Cb allows a single argument which can have any simple value (`int`, `char` or `string`). The method definition entered into the `System` namespace shows a single argument of type `object`. That method definition allows more than we want to handle in Cb (even though the additional types would be acceptable in C#), so your Cb compiler should report an error if the argument does not have a simple type.

Symbol Table Usage

You are provided with the code for a simple but very inefficient symbol table class in the file `CbSymTab.cs`. The functionality of the class is modelled after chapter 4 of Mogensen. To type check a method, you will need to perform the following steps:

- Create a new symbol table instance (or clear the contents of a previously allocated instance).
- Create bindings for each of the formal parameters of the method.
- When visiting the AST subtree for a Block, start a new scope before visiting the statements in that block, and exit the scope afterwards.
- When a declaration for a local variable is encountered, create a binding for it. The `Binding` method returns an instance of class `SymTabEntry` to hold that binding; the caller should insert type information into that instance.

- When a use of an identifier with no preceding dot is encountered, look it up in the symbol table. If the result comes back as `null` (not found), then you have to continue the search amongst the members of the current class. And if there is no member with the right name, then look amongst the members of the parent class, and its parent class, all the way up to the `object` class. Finally, you can check the current namespace to see if the name is there as a top-level name.
  Note: if the result of a look up is a field or a non-static method in the current class, then you must report an error if you are currently type checking a static method. (Cb only has non-static fields.)

Note that the C# language does not permit you to declare a local variable which has the same name as variable in an enclosing block or as a parameter of the current method. The `Binding` method in the supplied `SymTab` class enforces that rule.

The Provided Materials

- The zipfile `SuppliedMaterials.zip` holds the source code files for a solution to Assignment 2 plus an additional file `CbSymTab.cs` (which is currently not accessed by any of the other code).

Submission Requirements

1. You must provide all source files needed to build your program. Do *not* submit any files generated by `gplex` or `gppg`.
2. We will run gplex on your `.l/.lex` file, gppg on your `.y` file, and then compile all the C# files with the command `csc *.cs`. If these steps do not yield an executable program, you lose points.
3. You must combine all your files including a README file into a single compressed archive file. The only accepted formats for the archive file are as a zipfile (and the filename must have a "`.zip`" suffix) or as a gzipped Unix tar file (and the filename must have a "`.tgz`" suffix).
4. Upload your archive file via conneX.
5. The project is to be completed in teams of either 2 or 3 persons. The ideal size is 2 people. All team members *must* participate. Be sure to identify who the team members are in a separate file named `README.txt`. (If there are any special features of your code to point out, or things you didn't get working right, you can mention them here.)