CSC435/535: Assignment 3
(Due: 4 November 2012)

Introduction

This assignment asks you to fully type check and semantically check the AST constructed for a CFlat program.

Assignment Description

Type and semantic checking must be performed by the `CbTcVisitor` class. A template for this class plus a few fragments of code to get you started are provided.

Type checking means that every node in the AST which has a datatype must be annotated with that type (by setting its `Type` field to an instance of the `CbType` class). Any situation where an operator or a program construct has not been provided with an operand or subtree of the appropriate type must generate an error message. The error message must include the line number where the error was discovered.

Semantic checking mostly means checking that identifiers are used appropriately. Local variables must be declared before they can be used (but this does not apply to struct declarations, method declarations of const declarations). Variables or struct names or method names or const names cannot be declared twice in the same scope.

Additional semantic checking includes verifying that a return statement does not return a value in a void method (or return nothing in a non-void method). However it is tricky to verify that every control flow path returns a value in a non-void method, and you are not expected to include that check.

Semantic checking also includes checking that the only *using* identifier is `CbRuntime`, and that the method calls to perform input, output or obtain an array length use the correct identifiers. Refer to the CFlat Language Specification document for the semantic rules. (Any missing requirements can be inferred from the C# language.)

Overview of the Checking Strategy

A language like Java or C# cannot be type-checked in a single pass over the code. The problem is that some constructs, such as methods and structs, can be used before they are declared. The approach which has been partially completed for you in the supplied template for `CbTcVisitor` uses three passes.

The first two passes are separated out into a method named `prePass`, and they do not use the *Visitor* pattern for traversing the AST. Since these are special-purpose simple traversals which do not go deep down in the AST, they have been hard coded. The first pass examines all the declarations which are at the top level of the program (which is a single static class), and picks out all the *struct* type declarations. Each such declaration causes an entry to be added to a dictionary named `Structs`.

Now that all the struct declarations have been discovered, we know all the identifiers which are used as type names. The second pass goes over those declarations again, this time entering complete descriptions for each const, struct or method into three dictionaries (`Consts`, `Structs`, `Methods`).

Finally, now that all the names which can be used before they have been declared are entered into the three tables, a third and final type checking pass can commence. This checking should be implemented with the Visitor pattern. The template contains the methods and switch statements needed to give you the structure of the visitor.

While visiting nodes in this third pass, the normal coding approach for a visit to a non-leaf node N has this structure:

- Visit any children which need type-checking (by calling the `Accept` methods of these children). Note: if a child represents a datatype (it corresponds to the *Type* non-terminal in the grammar), you should not perform a regular type-checking visit to that node. Instead, call the method `lookUpType` which has been programmed for you already.
- Access the `Type` fields of any children which have just been visited, and verify that they are all compatible with each other and with the construct represented by the N node.
- Set the `Type` field of node N and return from the visit.

A visit to a leaf node, such as a string constant or identifier, is straightforward. If it's a string constant or a number, the `Type` field can be set directly. If it's an identifier, then it must be the name of a local variable (involving a symbol table look-up) or, if not found in the symbol table, it can only be the name of a constant which can be looked up in the `Consts` dictionary.

Programming Suggestion – Partial Classes

The `CbTcVisitor.cs` file is liable to become ridiculously large as you add all the type-checking code to it. You might like to consider splitting the `CbTcVisitor` class across two or more files. This is easily possible in C# through use of a feature known as *partial classes*. For example, if class B is a subclass of class A and needs to be split, we might split it into two files like these:

```
// File B1.cs
using System;
namespace Foo {
  partial class B:A {
    ... // some members of B
  }
}
```

```
// File B2.cs
using System;
namespace Foo {
  partial class B:A {
    ... // other members of B
  }
}
```

The Provided Materials

- You are given everything needed to start this assignment from scratch. If you wish, you may discard your solution to Assignment 2 and start over with the files provided on the conneX website. You are welcome (encouraged) to blend code, copying parts of the supplied files into your own implementation from Assignment 2. You may discover code in the supplied files which works better than yours – or *vice versa*!
- Note that the `CbType.cs` file has been rewritten. You should completely discard any copy of that file from Assignment 2 and replace it with this new version.
- The supplied source code files are listed in the table below.

File	Description
`cbc.cs`	The main program which invokes everything else. It has been updated to invoke the type-checking visitor.
`CbLexer.lex`	The specification file to be processed by gplex. It has been updated to support block comments.
`CbParser.y`	The grammar file to be processed by gppg. It has been updated to build the AST.
`CbAST.c`	The classes used for building the AST. (Unchanged.)
`CbVisitor.cs`	The parent class for the Visitor pattern. (Unchanged.)
`CbPrVisitor.cs`	A visitor for printing the AST. (Unchanged.)
`SymTab.cs`	A symbol table class for handling local variable declarations while type-checking a method body. (A new file.)
`CbTcVisitor.cs`	The template for a visitor for type-checking and semantically checking the AST. (A new file.)
`CbType.cs`	Classes used for describing Cb datatypes. This has been completely rewritten; please discard the Assignment 2 version.
`AST-DataStructure.pdf`	Explanation of the AST structure. (Unchanged.)
`AST-TypeRepresentation.pdf`	Overview of the structures used to describe CFlat datatypes.

- For Windows users, three batch command files are provided:

`rungplex.bat`	Runs gplex on `CbLexer.lex`
`rungppg.bat`	Runs gppg on `CbParser.y`
`build.bat`	Builds `cbc.exe` from all the C# source files

Linux or Mac OS X users can copy the commands into a Makefile or into shell scripts.

Submission Requirements

1. You must provide exactly one file. It must be a zip file or gzipped tar file which contains all the source code files needed to build your program. If you added more C# files to your project, include those too.
2. Also include a file named `README.txt` which identifies the team members. If you have any comments you want to share about problems with the assignment, this is an appropriate place to supply the comments.
3. Important: do *not* include any files generated by gplex or gppg in your submission, we want the `.lex` and the `.y` files to be submitted.
4. The project is to be completed in teams of either 2 or 3 persons. The ideal size is 2 people. The teams do not have to contain the same members as for Assignment 2.

Final Note

No guarantee is provided (or can be provided) that the supplied code is 100% correct. You become responsible for all the code which you include in your compiler project.

If you find any errors or deficiencies in the supplied code, please report them and they will be fixed as soon as possible. A separate document named `Corrections.txt` will then appear in the assignment 3 folder on conneX which describes the problem and its correction. That file will grow as needed. No special announcement will be made unless it is a serious problem that you would have trouble fixing without assistance.