CSC435/535: Assignment 2
(Due: 11:55pm, Sunday 14 February 2016)

Introduction

This assignment asks you to complete the programming of a visitor which traverses the parse tree and builds a symbol table for the top level (the package level) of a Goo program.

Assignment Description

- You are provided with several source code files. These are the Antlr files for Goo (which effectively include a solution to Assignment 1), Java files for symbol table support, for Goo type descriptions, and for an incomplete visitor.
  You may choose to discard your own solution to Assignment 1 and start over, or transfer the parts of the `Goo.g4` file which are needed to support Assignment 2 into your own version of the file. Either way is OK, except that you need to be reasonably sure that the `Goo.g4` used in this and future assignments does not contain errors.]
  The supplied version of `Goo.g4` comments out a few production rules in order to reduce the complexity of the Goo compiler. (For example, functions which return more than one value as their result will not be supported.) Also, some groups of production rules which used recursion to specify lists of items have been replaced with the Kleene start because that simplifies traversal if the parse tree by our visitor programs. A few other rules have been modified so that semicolons can now be omitted when the next token is a right parenthesis or a right curly bracket. (With this change, all the semicolon omission rules described in the Go Language Specification are now supported.)
- The supplied visitor code is a file named `SymTabVisitor1.java`. It must be extended so that it visits all the parse tree nodes corresponding to the non-terminal symbol `topLevelDeclList` in the grammar and enters the declared symbols into the symbol table. The visitor must visit the formal parameter declarations of each function so that the datatypes of the arguments are used to construct a function signature (and the signature is the datatype of the function). However the names of the formal parameters are ignored and the body of the function is not visited.
  [Assignment 3 will construct one or more visitors to process function bodies, performing full type checking and semantic checking.]
- The code for printing the contents of the symbol table calls a method in the `Symbol` class named `getLineNumber()` to obtain the line number where the symbol was declared. This method is unimplemented; it always returns 0. A minor task in this assignment is for you to implement that method. (Note that a line number of 0 is appropriate when a predefined symbol is displayed.)
- The supplied code contains methods for printing the contents of the symbol table when the option `-dsym` is provided on the command line. The desired results for some Goo language examples are shown below. Please do not change the format of the ouput! We can test submissions much more easily if every team uses the same format. If you prefer a different format (or desire additional output to be generated), feel free to define additional methods for producing the output and define an additional command line option which selects your own output format
- Note that the example in Figure 4 shows an interesting feature of the Go language which requires careful implementation to be supported in Goo.

Examples

Each example is shown as a pair of boxes. The first box contains the Goo program; the second box (lightly shaded) shows the output that the Goo compiler generates when invoked with the `-dsym` command line flag.

Figure 1: Example with functions

```
package main

func add(x, y int) int { return x + y }

func main() { println(add(42, 13)) }
```

```
package level names {
   3: Function add(int,int):int
   5: Function main()
}
```

Figure 2: Example with a struct

```
package main

type Books struct {
    title string; author string; subject string; book_id int }

func main() {
    var Book1 Books
    Book1.title = "Go Programming"
}
```

```
package level names {
   3: TypeName Books:struct{ [Field title:string, Field
author:string, Field subject:string, Field book_id:int] }
   6: Function main()
}
```

Figure 3: Example with two structs

```
package main

type BookInfo struct { subject string; book_id int }

type Books struct { title string; author string; info BookInfo }

func main() {
    var Book1 Books
    Book1.title = "Go Programming"
}
```

Figure 3: Example with two structs (Continued)

```
package level names {
   3: TypeName BookInfo:struct{ [Field subject:string, Field
book_id:int] }
   5: TypeName Books:struct{ [Field title:string, Field
author:string, Field info:struct{ [Field subject:string, Field
book_id:int] }] }
   7: Function main()
}
```

Figure 4: Example with use before definition issue

```
package main

type Books struct { title string; author string; info BookInfo }

type BookInfo struct { subject string; book_id int }

func main() {
    var Book1 Books
    Book1.title = "Go Programming"
}
```

```
package level names {
   3: TypeName Books:struct{ [Field title:string, Field
author:string, Field info:struct{ [Field subject:string, Field
book_id:int] }] }
   5: TypeName BookInfo:struct{ [Field subject:string, Field
book_id:int] }
   7: Function main()
}
```

Figure 4 illustrates an important issue for your implementation. Line 4 of the Goo program refers to a symbol, `BookInfo`, which is not declared until later in line 7. The supplied code gives an error message saying that BookInfo is not a type name. However, this forward use of a symbol is actually not an error in either Go or Goo. You have got to make it work.

There is a standard solution used in compilers. When the symbol `BookInfo` is first encountered, it has not been declared in the current scope. You should add a definition for the symbol to the current scope (the package level scope), marking its kind as `TypeName` but setting its datatype to something which implies "not known yet". The supplied code for `Type.java` provides such a datatype value; it is `Type.unknownType`. Later, when the definition for `BookInfo` is encountered by the visitor, the visitor updates the datatype from *unknown* to the appropriate struct type.

One final detail is that the visitor must make a check when the end of the Goo program is reached that no type names at the package level scope are left with a datatype of *unknown*.

Commands for Building and Running the Goo Compiler

The following commands are recommended. They assume that you have previously created a new subfolder (subdirectory) named `bin` which will be used to hold all the class files created by the Java compiler. There will be more than 100 class files, so separating them from the source code is a good idea. The last argument on the last command is the path to the Goo source code program which is to be compiled.

```
% antlr4 -visitor -no-listener Goo.g4
% javac -g -d bin *.java
% cd bin
% java GooMain -dsym ../../progs/prog1.go
```

The use of two command windows, one whose current working directory is the folder holding the source code and another whose working directyory is the bin folder, will make rebuilding and rerunning the Goo compiler easier. Alternatively, use an IDE like Eclipse to manage and test the project.

The Provided Materials

See Table 1.

Evaluation Criteria

You will be evaluated on these criteria.

- Meeting the submission requirements.
- Handling all top level (package level) declarations.
- Handling the use before declaration issue for type names.
- Showing line numbers in the symbol table dumps.

Submission Requirements

1. You must combine all the source code files into a single compressed archive file and upload that to conneX. Do not include any class files generated by the Java compiler. (There will be more than 100 of them!). Any Java files generated by Antlr can optionally be included in the archive file, but they will be ignored. Please use a filename that begins '`ass2`' and has the appropriate suffix for the file format. The choices are: `ass2.zip` (Zip file), `ass2.tgz` (gzipped tar archive) or `ass2.rar` (for "Roshal ARchive format" according to Wikipedia).
2. The comment at the start of the `GooMain.java` file must be edited so that it lists the names and student numbers of the team members who collaborated on the assignment.
3. The submitted code must not produce any error messages or warning messages when processed by Antlr4 or when the Java files are compiled. Errors that prevent the main program from invoking the visitor will lead to an automatic zero.
4. Your program when invoked with the `-dsym` command line option must generate symbol table output in the same format as supported by the supplied code. *There must be no additional output.*

5. The project should be completed in teams of either 2 or, at most, 3 persons. Single member teams are permitted but not encouraged. Your team does not have to contain the same members as for Assignment 1.
6. Late assignments are accepted with a penalty: 10% (of maximum score) for up to 24 hours late, 25% for up to 48 hours late, 50% for up to 72 hours late. You can resubmit as often as you want. The time of the latest submission determines which late penalty, if any, is to be applied.

Table 1: Materials Provided in the Resources/Assignment2 Folder on ConneX

File Name	Description
`Goo.g4` `GooLexerRules.g4`	Two textfiles containing a complete Antlr4 specification for Goo. The lexer rules have been separated out into a second file which is imported by the first file. Significant changes in `Goo.g4` are tagged with a comment saying 'CHANGED'.
`Type.java`	Classes which represent the Goo datatypes. The parent class is named `Type`; definitions for several subclasses are provided as nested classes.
`Symbol.java` `FunctionSymbol.java`	`Symbol` is a class used to associate information with a symbol declared in the Goo program. `FunctionSymbol` is a subclass which allows additional information to be kept for functions.

`Scope.java`
`BlockScope.java`
`FunctionSymbol.java`
	The `Scope.java` file defines an interface for accessing one scope level of a tree structured symbol table. The interface is implemented by the classes defined in the two other files.

`Packages.java`	The packages class adds definitions of symbols imported from libraries into the top level symbol table scope. *The code for this class is incomplete*. (Not needed for Assignment 2.)
`Predefined.java`	The `Predefined` class adds definitions of Goo's predefined symbols to the global symbol table scope. Such symbols include `int32` and all the typenames, as well as some builtin functions.
`ReportError.java`	Static methods in this class must be used for reporting all error and warning messages from the Goo compiler.
`SymTabVisitor1.java`	The file contains an incomplete visitor for building the global scope and package level scope of the symbol table.
`GooMain.java`	A main class which performs all steps up to and including calling the visitor on the parse tree.