CS5300
Code Formatter

In this project you will implement a formatter for the so-called C- language. The grammar has been written for you. You will

- Compile the grammar into Java code using Antlr. This builds a parser for you. (Yay, you don't have to implement your own!)
- Build an abstract syntax tree (AST).
- Build a hierarchical symbol table.
- Print an error if a variable is used without having been defined.
- Traverse through the AST, outputting the code formatted nicely.

Setup

1. In this project you will submit only files in the `submit` package. You may modify other files such as **main.java**, but you will not be submitting them.
2. Load the Formatter project into IntelliJ or your choice of IDE. The `main()` method is in **main/Main.java**. After compiling the grammar (see below) you should be able to run it without errors, but it won't really do anything. Familiarize yourself with **Main.java**.
3. Make sure **antlr-4.9.1-complete.jar** is in your classpath. (If you added the code into IntelliJ using *New > Project from existing sources...* then it should have added the jar file automatically.)

Compile grammar

1. Find **grammar/Cminus.g4**. This is the grammar for C-. You will be looking at this file a lot.
2. At the command-line in the **grammar** directory, run
   ```
   java -jar ../antlr-4.9.1-complete.jar \
          -o ../parser -package parser Cminus.g4 -visitor
   ```
   This will generate Java files that will parse C- programs. The files are written to the **parser** directory which is sibling to the **submit** directory.
3. You should now be able to compile and run the program.

Build AST

You will build an abstract syntax tree (AST) from the parse tree by traversing the parse tree and creating AST nodes as you go. For traversal, we will use ANTLR-provided code in the form of a visitor. In the visitor pattern, we write a "visitor" class and ANTLR's traversal code calls methods on our visitor indicating what parse tree nodes it is visiting.

1. Look at the `ASTVisitor` class. You will be doing much of your work here. You will also be adding classes to the `ast` package.
2. Start overriding functions in your visitor class that create AST nodes. A couple of examples are given, and some AST nodes are already created for you. See how the methods correspond to the production rules in **Cminus.g4**.
3. As you look at **Cminus.g4**, notice that ANTLR syntax replaces a typical recursive grammar rule

   ```
   andExpression : andExpression '&&' unaryRelExpression | unaryRelExpression ;
   ```

   with the much simpler, iterative

   ```
   andExpression : (unaryRelExpression '&&')* unaryRelExpression ;
   ```

   This makes traversing the parse tree far simpler.
4. You will format your code to match what is given in **testx.out**.
5. As you go, start your testing on **data/test0.c**. Once that is complete, test on **data/test1.c**.
6. Each `Context` object has a `getChild(int index)` method that can come in useful. They also each have a `getText()` method that will also be helpful.
7. The default implementation of the `visit...()` methods is to call `visitChildren()`. In some cases you may create an AST node or two and then call `visitChildren()`, but in most cases (such as an `if` statement) you may want more control over how the children are called. In these cases you can call `visit...()` directly on the children as needed.
8. Use `LOGGER.fine()` for debugging statements.

Symbol table and error reporting

1. As you traverse through the parse tree and build the AST, build your hierarchical symbol table. Two classes, `SymbolInfo` and `SymbolTable`, are included that you can use if you like.
2. To populate the symbol table, add a symbol every time a variable or function is declared, and with every function parameter (a function's parameters will be the first entries into the function's symbol table).
3. To check for errors, check the symbol table every time a variable is used or a function is called. If I had a MutableContext called `mc` and the ID wasn't found in the symbol table, then I would print an error as follows:

   ```
   LOGGER.warning("Undefined symbol on line " + mc.getStart().getLine() + ": " + id);
   ```

Formatter

1. Notice that most AST node classes have a method called `toCminus`. This method adds formatted code to `builder`. Notice that `Program.toString()` calls these functions. As you add AST node classes, make sure each one has a `toCminus` method.
2. When you run your formatter on **data/testx.c** you should get results identical to what is found in **data/testx.out**.
3. Your output must match **data/testx.out**. *You should have no extra output beyond what is in this file.* For debugging purposes, you may want to add debug output. To do this, use `LOGGER.fine(msg)` and make sure `trace=true` in the `main` class. To add output to match the desired output, use `LOGGER.info(msg)`.
4. Note that the desired bracing style in the formatter is different from what we use in this class. This is because it is easier to automatically format with opening braces on their own line.

Submission

1. Set `trace=false` in the `main` class and test your code to make sure you're getting the correct output without debug output.
2. Zip your **submit** directory and submit it. Zip only that directory! Make sure to zip the *directory* and not just the *files* in the directory. You will submit a single file called **submit.zip**.

Scoring

1. 5% - Compiling grammar with Antlr and importing code
2. 30% - AST
3. 25% - Formatted output
4. 30% - Symbol table and error reporting
5. 10% - Coding quality and style