CS4481
Project 6 - 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
- 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. Load the Formatter project into your IDE. You should be able to run it on **data/test1.c** (passed as a commandline parameter) without errors (but it won't really do anything). Familiarize yourself with **Formatter.java**.
2. If you haven't already installed Antlr, use the instructions in project 2 to install it.

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, run
   ```
   java -jar [...]/antlr-4.5.1-complete.jar
         -o parser -package parser Cminus.g4
   ```
   where [...] is the path to your antlr jar file. Within the directory from which you run this command, a directory called "parser" will be created containing Java files that parse C- programs. Note what this has done: it has "written" a Java program that is a compiler for the specific language that was given in the input file (i.e., C- programs).
3. Copy **parser** to your **src** directory (you may need to refresh your IDE).

Build AST

1. Copy the following code into your `format` method.

   ```
       ANTLRInputStream chars = new ANTLRInputStream(new FileReader(filename));
       CminusLexer lexer = new CminusLexer(chars);
       CommonTokenStream tokens = new CommonTokenStream(lexer);
       CminusParser parser = new CminusParser(tokens);
       parser.setBuildParseTree(true);
       ProgramContext programCtx = parser.program();
   ```

   `programCtx` is the root node of the **parse tree** that is built by Antlr as a result of parsing the file passed in to the commandline (i.e., data/test1.c).
   Remember that while parse trees can detect errors in syntax, they cannot tell, for example, whether a variable has or has not been previously declared before it is used. For this purpose we need to build a (hierarchical) `SymbolTable` to test whether each variable has been declared or not. This is one of the goals of this project.
   Before worrying about the `SymbolTable`, however, the primary, initial goal is to create an abstract syntax tree from the parse tree that you are given (see section 5.2 in the text for a review of ASTs). You should be sure that your AST is working completely before trying to worry about using the Symbol Table to detect declaration errors.
   Note that `programCtx` has a method `List<DeclarationContext> declaration()`. This returns a list of all declarations in the parsed program (hence the '+' in the `program -> declaration+` rule in **Cminus.g4**).

2. Just to get you familiar with the what is in `programCtx`, try the following code. Get very familiar with it and understand exactly what it does. See how it corresponds to the production rules in **Cminus.g4**. Make changes and see what happens. The project will go much more smoothly if you do. As a litmus test for how well you understand the code, what is the purpose of the null-check in the following code? *Note:* be sure to add the -v option when testing so that you see the `fine` statements from `LOGGER`.

   ```
       DeclarationContext declarationCtx = programCtx.declaration().get(0);
       VarDeclarationContext varDeclarationCtx = declarationCtx.varDeclaration();
       if (varDeclarationCtx != null) {
         String type = varDeclarationCtx.typeSpecifier().getText();
         List<VarDeclIdContext> ids = varDeclarationCtx.varDeclId();
         for (VarDeclIdContext id : ids) {
           LOGGER.fine(type + " " + id.getText());
         }
       }
   ```

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. Once you have some understanding of how the parse tree works, start setting up your AST nodes and populating them with values from the parse tree. Note you will have one class for each possible type of AST node. What children each type of node has depends on the type of node that it is (hence why you will be looking at the grammar file a lot). I strongly recommend putting all AST node files in a separate package called `ast`. Note the many of the classes for nodes already exist in `ast`. These classes are just to get you started. You will

add more classes such as `FunDeclaration`, `CompoundStatement` and others (which each correspond to grammar symbols).

5. Gradually populate your AST nodes with functions similar to the following:

```
public Program buildProgram(ProgramContext programCtx, SymbolTable symbolTable) {
  this.symbolTable = symbolTable;
  List<Declaration> declarations = new ArrayList<>();
  for (DeclarationContext declarationCtx : programCtx.declaration()) {
    if (declarationCtx.varDeclaration() != null) {
      VarDeclarationContext vdc = declarationCtx.varDeclaration();
      declarations.add(getVarDeclaration(vdc, false));
    } else if (declarationCtx.funDeclaration() != null) {
      FunDeclarationContext fdc = declarationCtx.funDeclaration();
      declarations.add(getFunDeclaration(fdc));
    }
  }
  Program program = new Program(declarations);
  return program;
}

private VarDeclaration getVarDeclaration(VarDeclarationContext varDecl, boolean isStatic) {
  VarType type = VarType.fromString(varDecl.typeSpecifier().getText());
  List<String> ids = new ArrayList<>();
  List<Integer> arraySizes = new ArrayList<>();
  for (VarDeclIdContext idCtx : varDecl.varDeclId()) {
    if (idCtx.NUMCONST() != null) {
      ids.add(idCtx.ID().getText());
      arraySizes.add(Integer.parseInt(idCtx.NUMCONST().getText()));
    } else {
      ids.add(idCtx.ID().getText());
      arraySizes.add(-1);
    }
  }
  VarDeclaration vd = new VarDeclaration(type, ids, arraySizes, isStatic);
  for (String id : ids) {
    this.symbolTable.addSymbol(id, new SymbolInfo(id, type, false));
  }
  return vd;
}
```

   I recommend that you have these functions in a separate class (I created a class called `ASTBuilder`).

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.
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 (one of the symbols in the grammar) 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/test1.c** you should get results identical to what is found in **data/test1.out**. You can test one of two ways:
   (a) Choose Run > Clean and build project. Then at the command line:
   ```
   java -jar dist/Formatter.jar data/test1.c
   ```
   (b) Set the command-line parameter `data/test1.c` at Run > Set project configuration > Customize... and then run with F6.
3. Your output must match **data/test1.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 add -v to the command-line. To make things print to match the desired output, use `LOGGER.info(msg)`.
4. Note that the desired bracing style in the formatter automatically formats with opening braces on their own line.

Submission

When you are done, zip your entire project into project3.zip and submit using Moodle.

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