Compiler Project

1. **Introduction**

2. **Background**

A compiler is a program that takes a text file written in some language and creates byte-code for the written program that can be executed on some target machine.

The purpose of compilers is to allow programmers to develop using higher-level, more powerful languages that are to read and write than lower-level languages (I.E.: c or assembly).

Compilers are partitioned into 4 components to reduce coupling and simplify the compilation process. These 4 components are:

* Scanner
* Parser
* Symbol Table
* Code Generation

3. **Methods and Discussion**
    A. ANTLR Setup
    B. Scanner

A scanner (also known as a tokenizer or lexer) is a program that reads in a text file that is a series of characters representing a program of a specific language. The scanner then produces a series of tokens that are passed on to the parser to analyze. A token is the most basic of components of a programming language. Characters are separated into different types of tokens based on their functions (keywords, operators, identifiers, separators, and reserved words). These tokens are dependant on the rules of the specific programming language that is being used.

After installing and testing ANTLR, a partial grammar for a language called Little was created from the text file describing the structure of the language. The grammar file included rules for identifiers, keywords, and operators. Based on the grammar, ANTLR was used to generate the basics of the scanner in Java (see figure below). Java was decided upon based upon the common experiences of the group members.

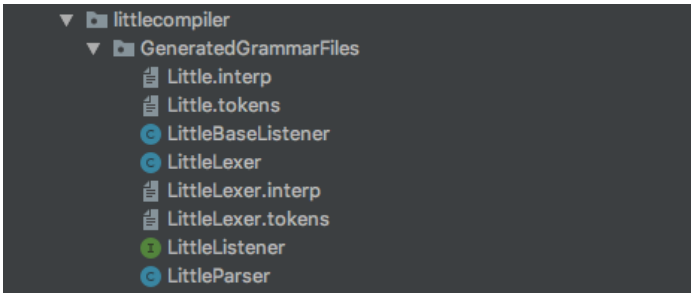

Figure: List of automatically generated files.

There were four files created in addition to those that were auto generated. The driver class was used as a start point. A factory class was written to help with dependency injection and to make unit testing of written files easier. The factory pattern was used to keep class coupling to a minimum. This type of coupling made the project more flexible and less fragile. The compiler class was written to aggregate all the tools used for the entire Compiler class project. The compiler class was created with two dependences, the lexer (or scanner) and the parser. The implemented method is for the scanner and was used to generate a file for holding the token output to be used later by the parser. The method for the parser was left as a stub since it was not needed for this part of the project. The last hand written class was token visualizer. This class was used to generate a string of all the tokens from a given program and then compared to a known output. There were five different programs used to test the operation of the scanner output and each difference was return with a value of zero. The two auto generated classes used were LittleLexer and LittleParser. These names were automatically given from the name of the grammar used and the function of the class. Lastly, script files were made to aid in running the appropriate files to recreate the ANTLR files and then use the hand written driver file to automatically scan test files for grading.

There were a few difficulties in accomplishing this first part of the compiler project. The first came from learning the nuances of the regular expressions that ANTLR will accept. For example, in the figure below the regular expressions for a string literal and use the tilde symbol (~) as NOT. This was not listed at either http://www.rexegg.com/regex-quickstart.html or https://www.shortcutfoo.com/app/dojos/regex/cheatsheet and was eventually arrived at through trial and error. So, ~[' '] means 'NOT nothing' or 'anything' is acceptable as a string literal and a comment.

```
6   D:          [0-9];              // digit
7   AN:         [a-zA-Z0-9];        // alphanumeric
8   ID:         [a-zA-Z]AN*;        // identifier
9   INTLIT:     D+;                 // integer literal
10  FLOATLIT:   D*.D+;              // float literal
11  STRINGLIT:  '"' ~['']* '"';     // string literal
12  COMMENT:    '--'~['']* '\n';    // comment
13  WS:         [ \t\r\n]+ -> skip; // skip spaces, tabs, carrage returns, and newlines
```

Figure: Identifiers from the grammar Little.g4

Another hurdle to get past was learning the ANTLR API. This took time to learn the new tool but was made a little easier from experience using other development environments and APIs. The automatically generated files were not exactly human readable and took a while to wade through along with searching for and reading through ANTLR documentation. This in turn helped to solve another challenge, how to pass files to the parser and then, in turn, pass tokens to the parser. The ANTLR libraries provided options and methods to help keep the hand written code simpler and easier to manage. The last item to work through was how ANTLR handled scanning and parsing. In class this was handled as two different but related steps in building a compiler. ANTLR, however, generates both of these components at the same time, which made it more difficult to distinguish what exactly need to be done for this part of the project.