**Compiler Project**

1. **Introduction**

// What is this project about? A high-level introduction

// Why you did this? Motivation?

// Summary of the project, what you achieved?

// Introduction to sections that follow.

1. **Background**

// What is a compiler? What is its purpose? What are its

// components?. How do these components work together?

// Some real­life examples.

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 four components to reduce coupling and simplify the compilation process. These four components are:

\* Scanner

\* Parser

\* Symbol Table

\* Code Generation

1. **Methods and Discussion** 
   1. **ANTLR Setup**

ANTLR is a program which generates a scanner and parser for the user given a config file (of the file format .g4). G4-files contain 2 sections: the lexer and the parser. This program is provided by ANTLR in the form of a jar named 'antlr-4.7.1-complete.jar'. This jar contains multiple programs, including one that we used to generate a lexer and a parser.

As part of the process of setting up ANTLR, we followed a tutorial on using ANTLR provided by ANTLR at www.antlr.org. The tutorial provided the following .g4-file named Hello.g4:

grammar Hello;

r : 'hello' ID ;

ID : [a-z]+ ;

WS : [ \t\r\n]+ -> skip ;

After downloading and placing ANTLR's 'antlr-4.7.1-complete.jar' file-path into our environment variables, we were able to invoke the 'org.antlr.v4.Tool' program in this jar with the Hello.g4 file and an arbitrary string as input. This program scanned the string using the scanner defined in Hello.g4, and it determined if it was syntactically correct according to Hello.g4's grammar.

* 1. **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 the 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.

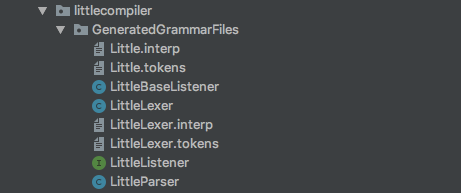


Figure: List of automatically generated files.

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.java and LittleParser.java. 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.

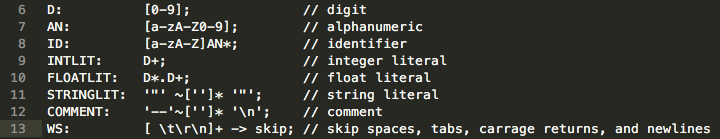


Figure: Identifiers from the grammar Little.g4

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.

* 1. **Parser**

According to Merriam-Webster the definition of parsing a grammar is “to divide (a sentence) into grammatical parts and identify the parts and their relations to each other” (https://merriam-webster.com). A parser is a program that acts as the interpreter part of a compiler that analyses a list of tokens by breaking them into smaller pieces and finds meaning in them that conforms to a set of rules defined by the grammar. Meaning can be made of these tokens by creating a parse tree of all of the tokens and their groupings. There are two stages involved in the parsing process: syntactical analysis, and semantic parsing. Syntactic analysis looks for a meaningful expression from the tokens generated by the lexer. The definition of the algorithmic procedures to create components comes from the usage of a context-free grammar. The placement of tokens must happen in a meticulous order and each of the rules in the grammar work to create an expression that has the proper syntax. This does not mean that the accepted string will have any meaning. The semantic stage of parsing determines the meaning and implications of a valid expression. There are two ways in which data from the input string of tokens can be derived from the start symbol: top-down parsing, and bottom-up parsing. Bottom-up parsing takes the entire list of tokens and tries to assemble a parse tree in reverse, working from the leaves to the start symbol. Top-down parsing, which is the way ANTLR works (see figure below), begins with the start symbol and works from left to right to find all the expressions that fit the given grammar. (https://www. techopedia.com)

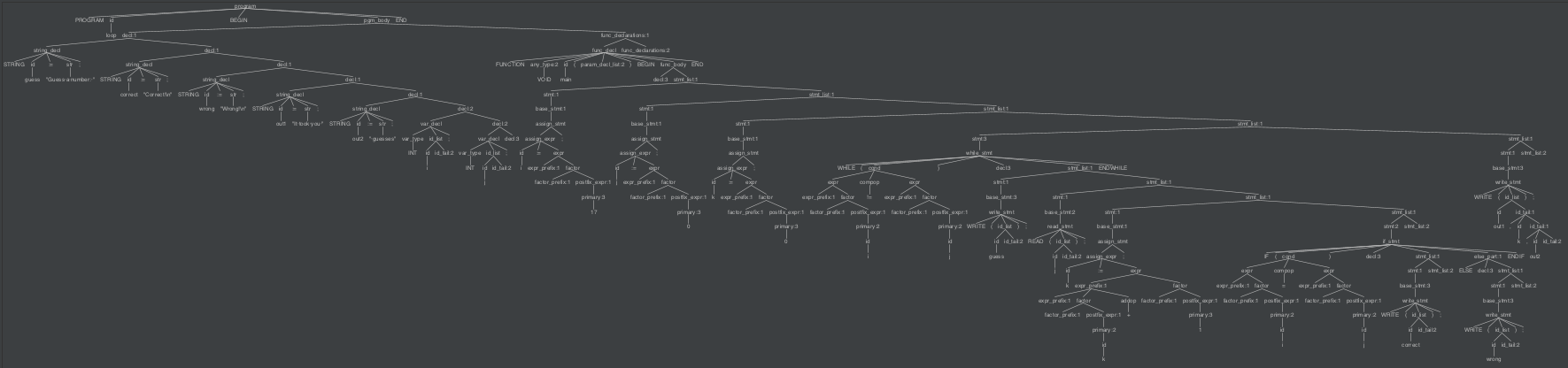


Figure: Generated parse tree.

There were a few changes to our code base for this stage of the project. First of all we needed to create a method generate a string of valid tokens that wasn’t printed out this time. This token stream was then to be fed into the parser as input. Before we could actually feed this stream to the parser, the LittleGrammar.g4 class needed to have all the productions added to it. This was a rather involved process since order and naming of non-terminals matters. A small sample of these productions can be seen below.

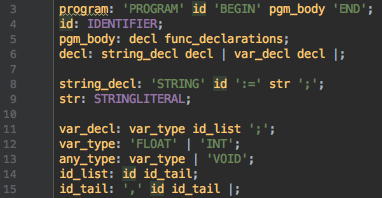


Figure: Sample of grammar productions.

We could then make the necessary addition to the driver to call the parser with the output string of tokens from the lexer. The sole output was to print that the input was either accepted or rejected. In order to accomplish this task we added a new class for error identification and handling. The factory class was also updated to accommodate this new dependency on the error strategy we selected.

Again, there were several roadblocks to overcome in finishing this part of the compiler project. First off, with the addition of productions to our grammar file, we found that there were needed changes to the original identifier list (seen in the figure below on the left). One identifier was added to the file and the names were changed to become more human readable. This also made it easier to add these to the different productions without needing to lookup what abbreviation belonged to each of the identifiers. The last change was in how to identify a comment. Originally a comment could contain anything in it and ended with a new line identifier (\n). This became problematic since the first new line encountered at the end of a line of code was turned into a comment. Simply excluding the new line symbol from the body of the comment identifier changed this undesirable behavior.

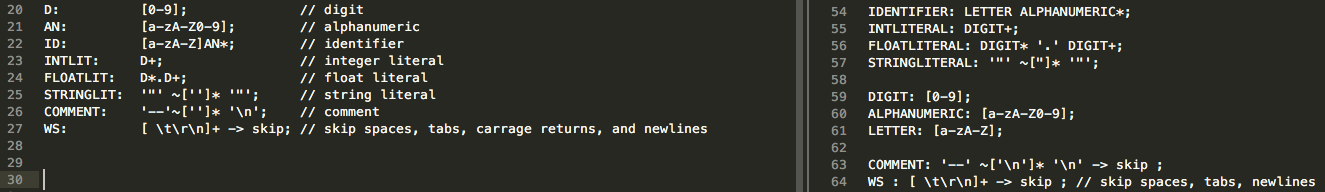


Figure: Changes to the identifiers in the grammar.

The next challenge came in trying to change the non-terminal names of some of the productions to make them more readable. However, this quickly cascaded into chaos and the only option was to go back to the naming convention given in the reference text file for the grammar. Even though it had been mentioned in class, we found that misordering productions caused havoc when trying to some of the test files. The challenge was trying to determine a proper order. This was mostly done by carefully reading through each of the productions and a little bit of trial and error. The last obstacle was perhaps the largest. In order to decide whether or not the input would be accepted or rejected we had to find some way to handle errors in parsing. There didn’t seem to be any way of catching errors since the lexer output a string of all tokens, valid or not. The parser would also work its way through the entire string fed to it even though there were marked errors in the output. After trying to catch these errors for both the lexer and parser and failing, it was decided to look for some hints in outside resources. This proved fruitful. The solution was to override some methods in one of the ANTLR libraries to do the error handling (https://stackoverflow.com/questions /39533809/antlr4-how-to-detect-unrecognized-token-and-given-sentence-is-invalid). Time to explore this solution was necessary to understand exactly what errors were being caught and how. The solution was finally modified to catch only the error we needed to either accept or reject the given input.