Compiler Project

2. Methods and Discussion

A. ANTLR Setup
B. Scanner/Lexer
C. 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)

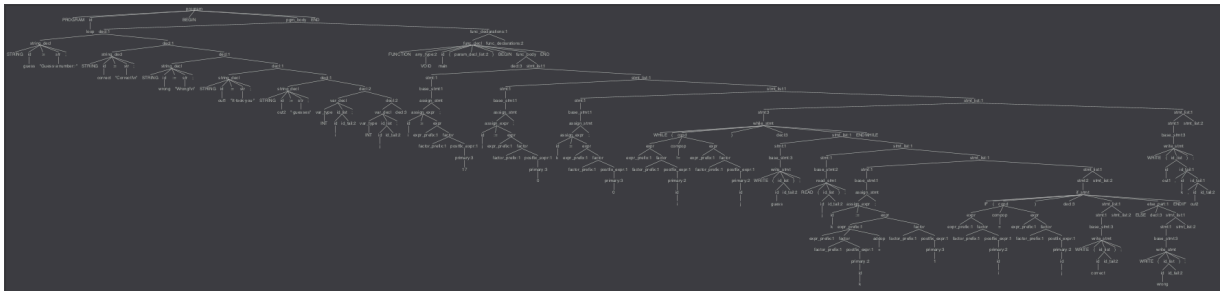

Figure: Generated parse tree.

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.

```
program: 'PROGRAM' id 'BEGIN' pgm_body 'END';
id: IDENTIFIER;
pgm_body: decl func_declarations;
decl: string_decl decl | var_decl decl |;

string_decl: 'STRING' id ':=' str ';';
str: STRINGLITERAL;

var_decl: var_type id_list ';';
var_type: 'FLOAT' | 'INT';
any_type: var_type | 'VOID';
id_list: id id_tail;
id_tail: ',' id id_tail |;
```

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.

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

```
IDENTIFIER: LETTER ALPHANUMERIC*;
INTLITERAL: DIGIT+;
FLOATLITERAL: DIGIT* '.' DIGIT+;
STRINGLITERAL: '"' ~["]* '"';

DIGIT: [0-9];
ALPHANUMERIC: [a-zA-Z0-9];
LETTER: [a-zA-Z];

COMMENT: '--' ~['\n']* '\n' -> skip ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

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.