Assignment Submission Form

Name(s): Gerard Slowey
Programme: CASE4
Module Code: CA4003
Assignment Title: Assignment 1 - A Lexical and Syntax Analyser
Submission Date: Monday 16th November 2020
Module Coordinator: David Sinclair

Name(s): Gerard Slowey Date: Saturday 14 November 2020

Table of Contents

Introduction

The aim of this assignment is to implement a lexical and syntax analyser using Antlr4 for a simple language called 'cal'. As part of the assignment I needed to build a parser that would parse a '.cal' file supplied as input via the command line.

Firstly, I defined the cal language's lexical and syntax descriptions, these are stored in the grammar file named `cal.g4`.

To generate the classes and runtime support to build a compiler from the grammar, I invoked Antlr4 on the grammar file by running the command `antlr4 cal.g4 -no-listener`. I included the `-no-listener` option as the listener classes are generated by default by Antlr4, however they are not needed for this assignment.

This would result in two java files (the Lexer and Parser files) along with interpreter and token files, all of which are attached to my assignment submission. To compile these Java files, I ran the command `javac *.java` in the same directory as the source files.

Once the Java files have been compiled, I am able to parse any input supplied. Input is supplied by passing a '.cal' file as an argument to the parser, for example `java Cal file_name.cal`. The result of the parsing operation will be returned in the terminal indicating the file name and either "parsed successfully" or "did not parse".

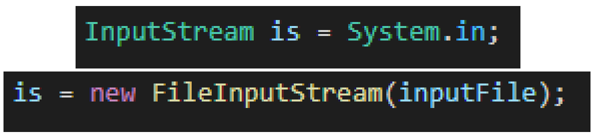
My main point of reference for developing my parser was the supplied course notes, pre-recorded videos and recorded zoom lab sessions, with the latest zoom lab session analysing the Boolean language being the most relevant in getting the parser structure correct.

Lexical Analysis

The goal of the lexical analysis part of this assignment is to convert streams of characters from the input file into streams of tokens. These tokens would represent keywords, identifiers, numbers and punctuation as defined in the grammar file.

The first step in this process is taking in the input file name from the command line. I included some error checking, for example making sure the file name was valid and that the name of the file supplied actually could be found on the system. This made the testing process less frustrating and helped prevent unexpected behaviour from the parser.

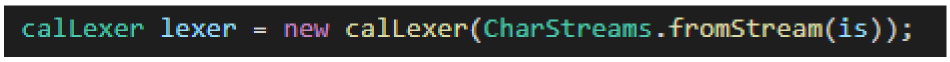
If an error occurred at these first stages of taking in the file, the program would prompt the user. If no errors occurred however, data would be read from the file in bytes and assigned to the 'InputStream' object **is**.



```
InputStream is = System.in;
is = new FileInputStream(inputFile);
```

Figure 1: Input File Parsing to Stream

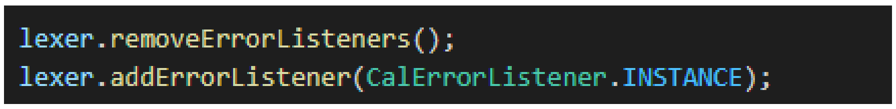
Using the generated java file `calLexer.java` I was able to create a 'calLexer' object called **lexer** and provide to it an input stream from the **is** object.



```
calLexer lexer = new calLexer(CharStreams.fromStream(is));
```

Figure 2: Lexer Extraction from File Stream

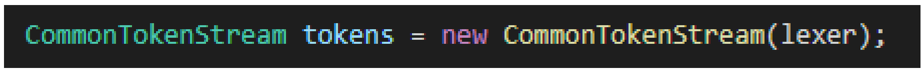
Before lexical analysis is performed, I first have to remove the default error listener. By doing this I was able to change the program behaviour when a lexical error occurs. Now instead of printing the lexical errors to the command line, an instance of the error listener 'CalErrorListener' is called and handles the error instead.



```
lexer.removeErrorListeners();
lexer.addErrorListener(CalErrorListener.INSTANCE);
```

Figure 3: Replacing Lexer Error Listener

After the lexer extracts the tokens they are assigned to a '`CommonTokenStream`' object **tokens**.



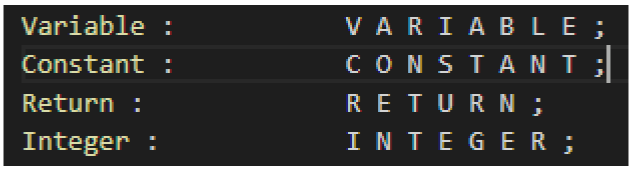
```
CommonTokenStream tokens = new CommonTokenStream(lexer);
```

Figure 4: Assigning Tokens to Token Stream

Grammar Lexical Structure

The main components of the grammar lexical structure are:

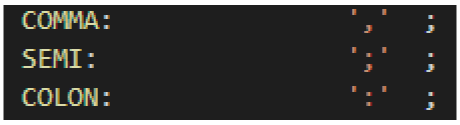
1. <u>Reserved Words</u>
   Reserved words are used internally in the grammar and take precedence over identifiers. Reserved words in this grammar are case insensitive, so fragments are used to match upper and lower case letters.



```
Variable :          V A R I A B L E ;
Constant :          C O N S T A N T ;
Return :            R E T U R N ;
Integer :           I N T E G E R ;
```

Figure 5: Example Grammar Reserved Words

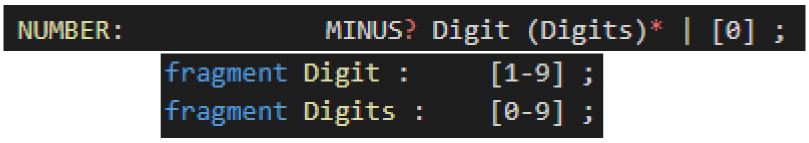
2. <u>Tokens</u>
   Tokens consist of symbols used by the grammar. Tokens are matched from a textual terminal description to the grammar symbol.



```
COMMA:                  ','   ;
SEMI:                   ';'   ;
COLON:                  ':'   ;
```

Figure 6: Example Grammar Tokens

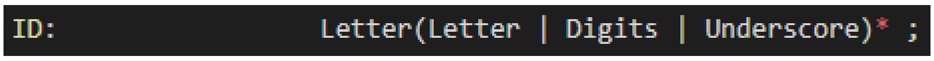
3. <u>Integers</u>
   Numbers in the cal grammar are represented by a string of one or more digits ranging from 0 to 9. Numbers are matched to token values given in the ranges below, otherwise 0 is matched. The number may be optionally negative (represented by ?) unless it is the number zero. Additionally, leading zeros are not allowed, meaning that an integer may consist of a digit ranging from 1 to 9, followed by zero or more numbers (represented by *) ranging from 0 to 9.



```
NUMBER:              MINUS? Digit (Digits)* | [0] ;
fragment Digit :       [1-9] ;
fragment Digits :      [0-9] ;
```

Figure 7: Handling Grammar Numbers

4. <u>Identifiers</u>
   Identifiers are represented by a string of letters, digits or an underscore character ‘ _’. The identifier must begin with a letter and cannot be a reserved word. Choice is represented using vertical bars between the options. Reserved words will precede identifisers, an example of a reserved word would be ‘variable’, whereas ‘variables’ would match an identifier.

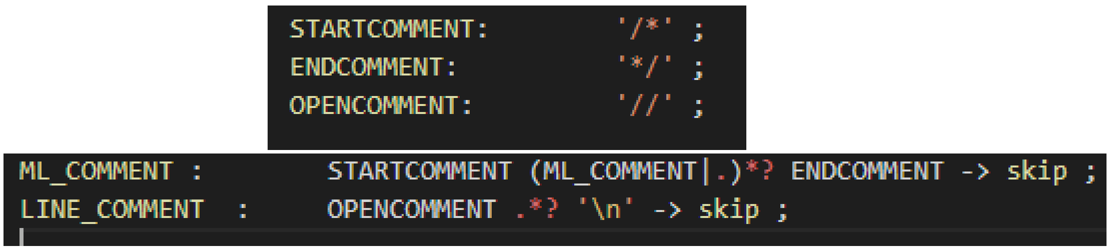


```
ID:                   Letter(Letter | Digits | Underscore)* ;
```

Figure 8: Grammar Identifier Structure

5. <u>Comments</u>
   There are two forms of comment, a multi-line comment is delimited by /* and */ and can be nested; the other, a single line comment begins with // and is delimited by the end of line and this type of comments may not be nested. Tokens are defined for opening and closing comment tags.

   Multi-line comments and single line comments use optionality (?) and repetition (*) tags to match comment formatting. To ensure the block is non-greedy the notation *? is used.



```
STARTCOMMENT:          '/*' ;
ENDCOMMENT:            '*/' ;
OPENCOMMENT:           '//' ;
```

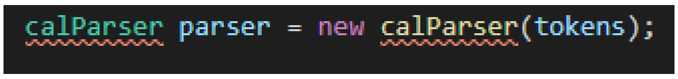
```
ML_COMMENT :         STARTCOMMENT (ML_COMMENT|.)*? ENDCOMMENT -> skip ;
LINE_COMMENT   :     OPENCOMMENT .*? '\n' -> skip ;
```

Figure 9: Grammar Comment Handling

Syntax Analysis

The goal of the syntax analysis stage of this project is to check whether the combined generated tokens form into a valid sentence that matches the grammar. In this grammar prog is designated as a start word.

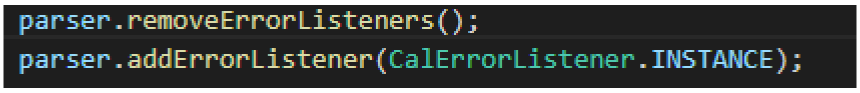
The syntax analysis process continues on from lexical analysis. Since the tokens have been extracted these can be supplied directly to the parser.



```
calParser parser = new calParser(tokens);
```

Figure 10: Passing Tokens to the Parser

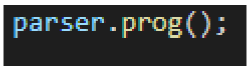
Similarly, as before I needed to remove the default error listener for the parser. This is again replaced by an instance of the CalErrorListener.



```
parser.removeErrorListeners();
parser.addErrorListener(CalErrorListener.INSTANCE);
```

Figure 11: Removing Parser Error Listener

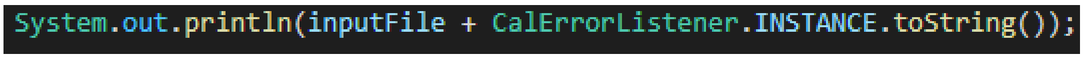
The parser is then run from the starting point of the grammar, `prog()`.



```
parser.prog();
```

Figure 12: Executing the Parser from the Prog Rule

Once the parsing has completed, the system will print out the result associated with parsing each individual file. The `toString()` method is called which returns the status message associated with the **CalErrorListener** instance.



```
System.out.println(inputFile + CalErrorListener.INSTANCE.toString());
```

Figure 13: Printing the Parsing Result

Generating a Parse Tree

We use the defined grammar to structure a parse tree given some input.

The main prog rule in the grammar states that prog follows this structure 'decl_list function_list main', however both 'decl_list' and 'function_list' can be empty, represented using the ? symbol after the grammar rule. By using the test function 'grun' with the '–gui' function selected and by supplying it with a test file, we can visualise this in a parse tree.

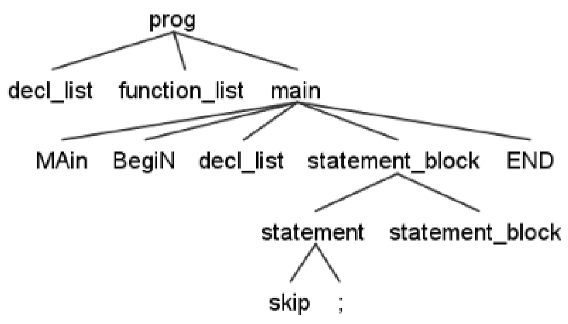


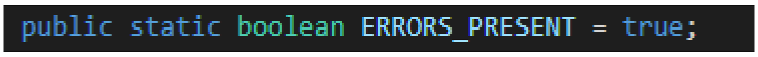
Figure 14: test7.cal Parse Tree

Error Handling

In order to implement the required textual result returned after a parsing or lexical analysis operation I needed to remove the standard error listener implemented by Antlr4. This allowed me to customise the error message to match the required "parsed successfully" or "failed to parse".

For guidance on this I used Chapter 9 "Error Reporting and Recovery" from the book "The Definitive ANTLR4 Reference, 2nd Edition". The most relevant examples featured are explained from pages 154 to 157, where the **syntaxError** function is overridden. I used this same idea in implementing my custom error listener 'CalErrorListener'. The CalErrorListener class is an extension of the BaseErrorListener class implemented by Antlr4.

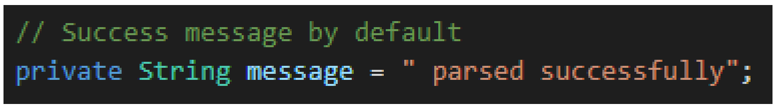
If the CalErrorListener is called, a Boolean variable `ERRORS_PRESENT` is set to true by default. This Boolean variable will change the final print statement and is altered based on the result of lexical analysis and parsing.



```
public static boolean ERRORS_PRESENT = true;
```

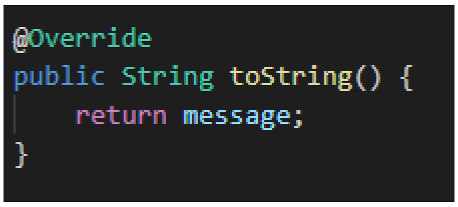
Figure 15: ERRORS_PRESENT Boolean

By default, if no errors occur during lexical analysis and parsing the ERRORS_PRESENT Boolean will be false, and the String type variable **message** is set to "parsed successfully". Otherwise, if an error has occurred, **message** is set to "has not parsed". The `toString()` method is called from the main `Cal.java` file and the value of the String message is printed to the console.



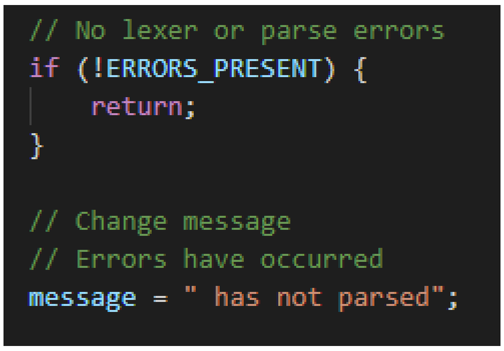
```java
// Success message by default
private String message = " parsed successfully";
```

Figure 16: Parse Success Message



```java
@Override
public String toString() {
    return message;
}
```

Figure 17: Error Listener toString Method



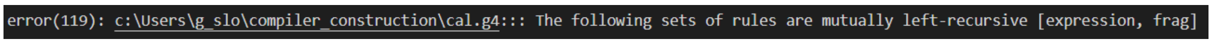
```java
// No lexer or parse errors
if (!ERRORS_PRESENT) {
    return;
}

// Change message
// Errors have occurred
message = " has not parsed";
```

Figure 18: Error Listener Boolean Check

Grammar and Production Rules

Encountering Recursion

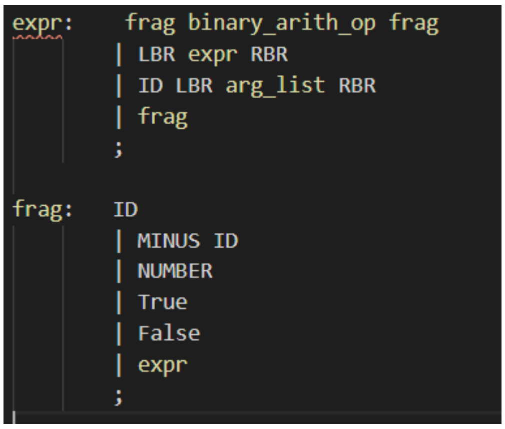
While defining the **expression** and **frag** nonterminals I encountered the following error:



```
error(119): c:\Users\g_slo\compiler_construction\cal.g4::: The following sets of rules are mutually left-recursive [expression, frag]
```

Figure 19: Recursion Error Message

Originally my **expression** and **frag** nonterminals were defined as follows:



```
expr:    frag binary_arith_op frag
       | LBR expr RBR
       | ID LBR arg_list RBR
       | frag
       ;

frag:  ID
       | MINUS ID
       | NUMBER
       | True
       | False
       | expr
       ;
```

Figure 20: Original expr and frag Grammar Structure

The recursive definition of expr and frag causes an infinite loop and prompts an error.

To solve mutual left recursion when defining the **expression** and **frag** nonterminals, I had to replace the invocation of rules, by the rule structure themselves.

The following link on stack overflow helped me with the transformation process.

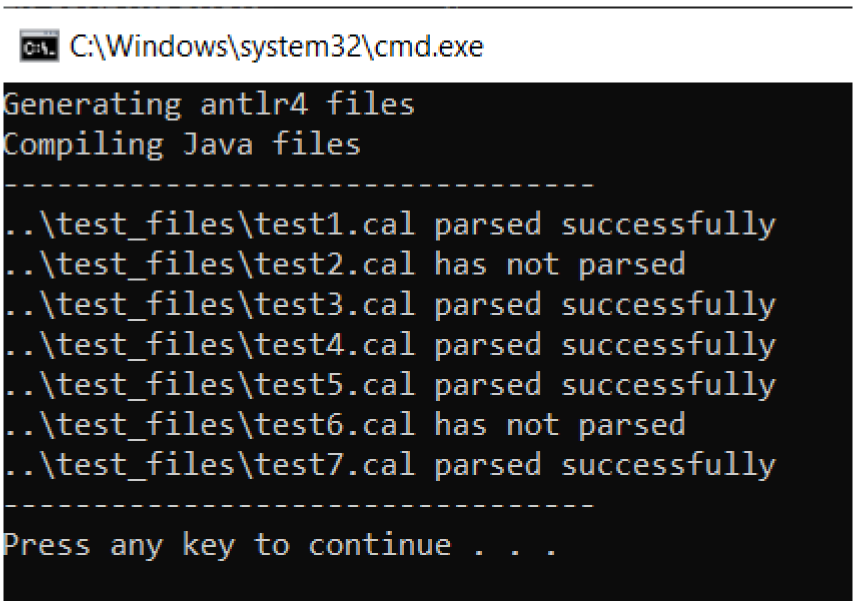
2014. *ANTLR4 Mutually Left-Recursive Error When Parsing*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/26460013/antlr4-mutually-left-recursive-error-when-parsing/26469983> [Accessed 1 November 2020].

Testing

The use of a batch file to run antlr4 and the grun test suite inspired me to use a batch file for testing the parser also.

To test the parsing and lexical analysis of the program I wrote a simple batch file named `testingCal.bat` to automate invoking antlr4, compiling the java files and supplying the test files to the parser one by one. Test files were supplied from a different directory hence the '..\test_files\' before each test file name.

The results of parsing can be seen here:



```
C:\Windows\system32\cmd.exe
Generating antlr4 files
Compiling Java files
---------------------------------
..\test_files\test1.cal parsed successfully
..\test_files\test2.cal has not parsed
..\test_files\test3.cal parsed successfully
..\test_files\test4.cal parsed successfully
..\test_files\test5.cal parsed successfully
..\test_files\test6.cal has not parsed
..\test_files\test7.cal parsed successfully
---------------------------------
Press any key to continue . . .
```

Figure 21: Testing .bat File Result

Conclusion

Part 1 of this assignment has given me a new appreciation for the underlying process of describing a grammar and rule matching. Antlr4 overall proved to be a very powerful tool in helping to produce a parser. This assignment has help me gain a better understanding of lexical and syntax analysis which should benefit me in the future both in this module and in code production generally.

I found the **gui** testing option to be very helpful in visualising a result and matching this to token values. Overall translating between the grammar declaration into the grammar file was challenging but allowed me to simplify certain grammar rules, avoiding repeated. By starting on this assignment in good time I was able to refactor certain elements as my understanding of concepts developed.