A Lexical and Syntax Analyser for the CCAL Language

CCAL.g4 is the grammar file that I implemented. The first thing I defined were the fragments which allowed for any case (upper or lower case) as CCAL is not case sensitive. Fragments are reusable parts of lexer rules and need to be referenced by a lexer rule.

```
fragment A:     'a'|'A';
fragment B:     'b'|'B';
fragment C:     'c'|'C';
fragment D:     'd'|'D';
fragment E:     'e'|'E';
fragment F:     'f'|'F';
fragment G:     'g'|'G';
fragment H:     'h'|'H';
fragment I:     'i'|'I';
fragment J:     'j'|'J';
fragment K:     'k'|'K';
fragment L:     'l'|'L';
fragment M:     'm'|'M';
fragment N:     'n'|'N';
fragment O:     'o'|'O';
fragment P:     'p'|'P';
fragment Q:     'q'|'Q';
fragment R:     'r'|'R';
fragment S:     's'|'S';
fragment T:     't'|'T';
fragment U:     'u'|'U';
fragment V:     'v'|'V';
fragment W:     'w'|'W';
fragment X:     'x'|'X';
fragment Y:     'y'|'Y';
fragment Z:     'z'|'Z';
```

I then moved on to define an Integer. According to the language definition, Integers are represented by a string of one or more digits (‘0’-‘9’) that do not start with the digit ‘0’, but may start with a minus sign (‘-’), e.g. 123, -456.

For this, I created a fragment called Integer and this can match a ‘0’ by itself or start with a minus sign followed by 1-9 (as we don’t want leading zeros), followed by 0 or more 0-9 or can match 1-9 followed by zero or more 0-9.

```
fragment Integer:       '0' | MINUS [1-9] ([0-9])* | [1-9] ([0-9])*;
//Can either have a 0 by itself or start with a minus sign followed by
1-9 followed by zero or more 0-9 e.g -234 or 1-9 followed by zero or
more 0-9 e.g. 567
```

Once I had integers figured out, I moved on to Identifiers. An Identifier can be a string of letters, digits or underscores but always starts with a letter. This can be seen here:

```
fragment Letter:      [a-zA-Z]+;
fragment UnderScore:     '_';
```

I first created a fragment “Letter” which matches lower and upper case a-z. I also defined an underscore in a fragment and it matches the underscore symbol ‘_’.

```
ID:     Letter (Letter | Integer | UnderScore)*;
//Identifiers are represented by a string of letters, digits or
underscore character beginning with a letter
```

I then created the Identifier rule (ID) which uses the fragments from above. An ID will always start with a letter but can also have 0 or more letters, integers or underscores.

The CCAL language also allows for comments, both line comments and nested comments.

```
COMMENT:    '/*' (COMMENT|.)*? '*/'    ->  skip;
```

This allows for comments using the /**/ symbols. Also allows for 0 or more tokens between the comment symbols. According to the language definition, these comments could allow nested comments. I implemented this by using recursion. As can be seen above, between both symbols we can have another comment or a token which is represented by “.”. This is a wildcard and can be replaced with anything. The “?” represents an optional argument. If the parser sees this, it can skip it.

I implemented a line comment as:

```
LINE_COMMENT:   '//' .*? '\n'    ->  skip;
```

Using “//” to start the comment followed by anything represented by “.” (wildcard) and finishes with a new line. This is then skipped.

The reserved words in the language are **var, const, return, integer, boolean, void, main, if, else, true, false, while** and **skip.**

I implemented these reserved words:

```
//Reserved words
VAR:    V A R;
```

```
CONST:  C O N S T;
RETURN: R E T U R N;
INTEGER:    I N T E G E R;
BOOLEAN:    B O O L E A N;
VOID:   V O I D;
MAIN:   M A I N;
IF:     I F;
ELSE:   E L S E;
TRUE:   T R U E;
FALSE:  F A L S E;
WHILE:  W H I L E;
SKIPS:   S K I P; //Had to change the word to 'SKIPS' because an error
occured otherwise e.g. "cannot declare a rule with reserved name SKIP"
```

I had to Change the reserved word “SKIP” to “SKIPS” as skip is already a reserved word in ANTLR.

I then created the tokens in the language as described in the language definition:

```
//The Following are tokens in the language
COMMA:  ',';
SEMI:   ';';
COLON:  ':';
EQUALS: '=';
CURLY_LBR:   '{';
CURLY_RBR:   '}';
LBR:    '(';
RBR:    ')';
PLUS:   '+';
MINUS:   '-';
NEGATION:    '~';
OR:     '||';
AND:    '&&';
EQUAL_TO:    '==';
NOT_EQUAL_TO:   '!=';
LESS_THAN:   '<';
LESS_THAN_EQUAL_TO:    '<=';
GREATER_THAN:    '>';
GREATER_THAN_EQUAL_TO: '>=';
```

All the other rules were defined in the language definition provided to me for this assignment.
The only rule that wasn’t defined was epsilon. I created a rule for that.

```
epsilon: ; //empty
```

It's a rule with nothing in it (empty).

I came across indirect left recursion in the rules "expression" and "fragment". In order to solve this, I read the Definitive ANTLR 4 Reference book. ANTLR 4 supports direct left recursion but doesn't support indirect left recursion. I struggled to figure out a solution but realised "expression" had already been defined using

```
            | LBR expression RBR
```

I decided to use this same implementation in "fragment" which solved the issue by placing terminals around the non-terminal.

```
frag:       ID
             | MINUS ID
             | NUMBER
             | TRUE
             | FALSE
             | LBR expression RBR
```

This seemed to clear up the issue but the test cases will also need to contain a "(" followed by an expression, followed by ")".

I tested the grammar using "grun CCAL prog -gui" and used the examples given in the language definition to check if there were any errors.

The language also has some right-associative operators. I researched this using the official ANTLR documentation and came up with a solution:

```
const_decl: <assoc=right> CONST ID COLON (INTEGER|BOOLEAN) EQUALS
expression;
```

```
statement:  <assoc=right> ID EQUALS expression SEMI
```

```
binary_arith_op:    PLUS
                    | <assoc=right> MINUS //arithmetic negation is
right to left associative according to the language definition
                    ;
```

We have to explicitly define when an operator is right associative as ANTLR by default is left associative. "<assoc=right>" is used to define a right associative operator and is placed on the left hand side before the operator.

CCAL.java file was similar to what was shown in the zoom lecture. Firstly, I imported the necessary libraries:

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import org.antlr.v4.runtime.CharStreams;
import java.io.FileInputStream;
import java.io.InputStream;
```

Then I created my CCAL class and main method. The CCAL language program can be taken in from a file on the command line.

```
String languageFile = null; //set the languageFile initially to null
        if(args.length > 0){//check if the command line arguments are
greater than zero
            languageFile = args[0]; //if there's arguments, let
languageFile equal the argument
            //our languageFile will be a program written in the CCAL
language

        }
```

I then created a stream that reads from standard input.

```
InputStream inputStream = System.in;
```

If there's a program written in the languageFile, our input stream will use this program.

```
if(languageFile != null){
            inputStream = new FileInputStream(languageFile);
        }
```

I then created a lexer.

```
CCALLexer lexer = new CCALLexer(CharStreams.fromStream(inputStream));
```

Using this lexer, I created the tokens.

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

Then, I created the parser that takes in the tokens

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

Finally, I started the parsing at the "prog" rule.

```
ParseTree tree = parser.prog();
```

If you want to see the tree, I found this line of code in the Definitive ANTLR 4 Reference book on page 26:

```
System.out.println(tree.toStringTree(parser));
```

This will display the tree on the command line. I have it currently commented out

References:

The Definitive ANTLR4 Reference book
- Learned more about ANTLR and how grammar's work using this book

Learned from the official ANTLR github by Terence Parr:
https://github.com/antlr/antlr4.

Learned about associativity in ANTLR here:
https://github.com/antlr/antlr4/blob/master/doc/left-recursion.md.

Learned about nested comments here:
https://stackoverflow.com/questions/27539351/how-to-handling-nested-comments-in-antlr-lexer.

How to remove indirect left-recursion from grammar:
https://stackoverflow.com/questions/19727458/how-to-remove-indirect-left-recursion-from-grammar.