Assignment 1

CAL (lexical and syntax analyser)

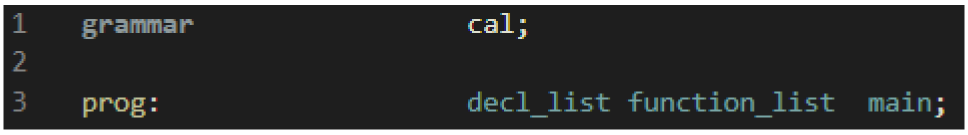
Name: Muhammad Umar Date: 15/11/2020

Introduction:

For this assignment I was supposed to write and create the lexical analyser for CAL Language as well as a syntax analyser to check if the file has parsed successfully or not. In this document I will explain the implementation of the lexical and the syntax analyser provided with some snapshots of the code.

Implementation(cal.g4):

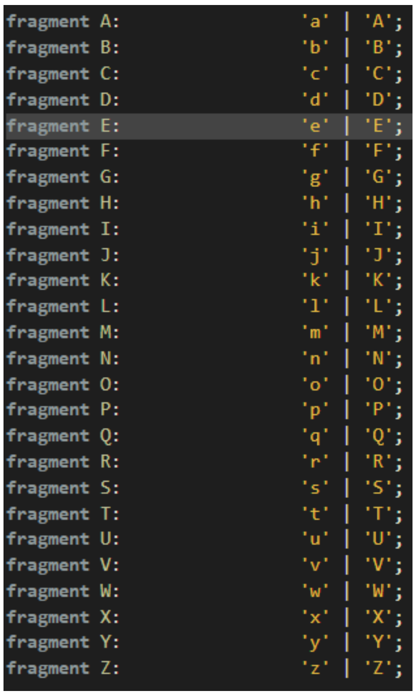
Starting with CAL as being the name for the .g4 file for which I will writing the rules, plus the prog as being what this file expects the user to input.



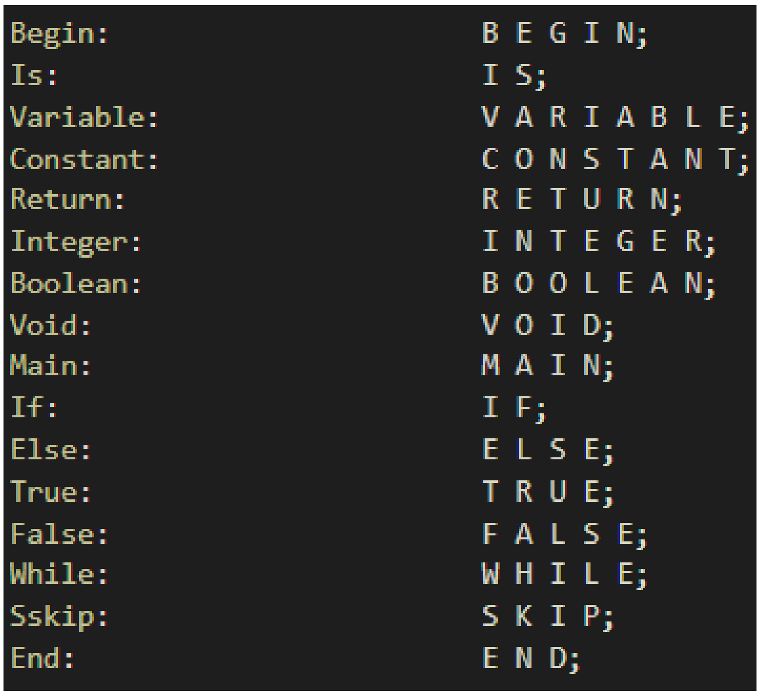
```
grammar                 cal;

prog:                   decl_list function_list  main;
```

CAL is not case sensitive, I first decided to write the fragments for each character and the Tokens which will be used as reserved words for this language. This will help me accept those tokens no matter if they lower/upper case. I have learnt this from one of the labs we did in which the language was not case sensitive also.

Reason for using 'Sskip' and not 'Skip' is because 'skip' is a reserved word in antlr4 and can cause confusion, so to keep things clear I did that.

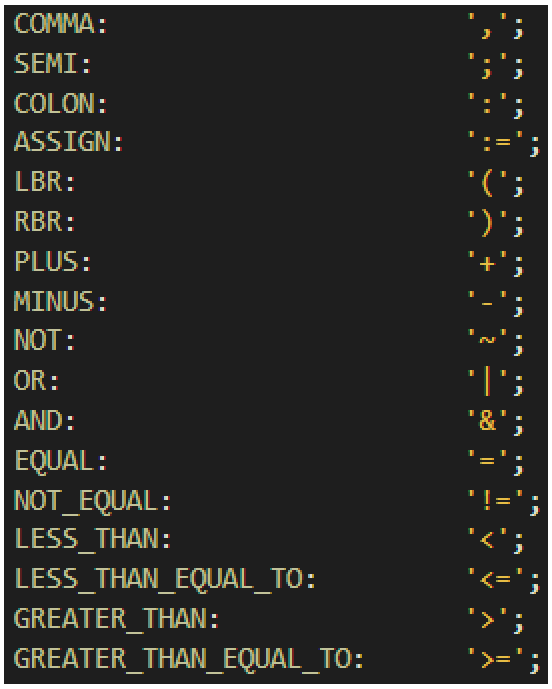


```
fragment A:           'a' | 'A';
fragment B:           'b' | 'B';
fragment C:           'c' | 'C';
fragment D:           'd' | 'D';
fragment E:           'e' | 'E';
fragment F:           'f' | 'F';
fragment G:           'g' | 'G';
fragment H:           'h' | 'H';
fragment I:           'i' | 'I';
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';
```



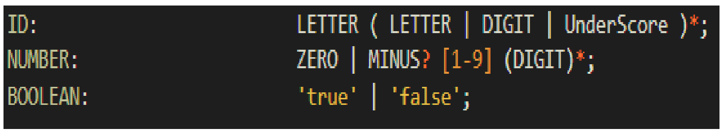
```
Begin:              B E G I N;
Is:                 I S;
Variable:           V A R I A B L E;
Constant:           C O N S T A N T;
Return:             R E T U R N;
Integer:            I N T E G E R;
Boolean:            B O O L E A N;
Void:               V O I D;
Main:               M A I N;
If:                 I F;
Else:               E L S E;
True:               T R U E;
False:              F A L S E;
While:              W H I L E;
Sskip:              S K I P;
End:                E N D;
```

Next, I also defined operators which were required for CAL.

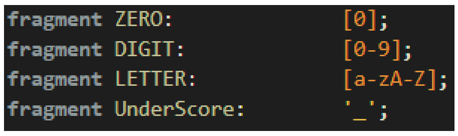


```
COMMA:                       ',';
SEMI:                        ';';
COLON:                       ':';
ASSIGN:                      ':=';
LBR:                         '(';
RBR:                         ')';
PLUS:                        '+';
MINUS:                       '-';
NOT:                         '~';
OR:                          '|';
AND:                         '&';
EQUAL:                       '=';
NOT_EQUAL:                   '!=';
LESS_THAN:                   '<';
LESS_THAN_EQUAL_TO:          '<=';
GREATER_THAN:                '>';
GREATER_THAN_EQUAL_TO:       '>=';
```

NUMBER was implemented using the help of DIGIT(0-9) & ZERO, as the rules says numbers shouldn't start with a '0' and number can contain a '-' negative number. I know this could have been implemented without the help of DIGIT & ZERO, but it looks neater this way. Identifier were also implemented this way.



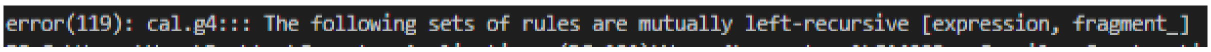
```
ID:                 LETTER ( LETTER | DIGIT | UnderScore )*;
NUMBER:             ZERO | MINUS? [1-9] (DIGIT)*;
BOOLEAN:            'true' | 'false';
```



```
fragment ZERO:              [0];
fragment DIGIT:             [0-9];
fragment LETTER:            [a-zA-Z];
fragment UnderScore:        '_';
```

The rest of the file contains rules which were taken from the pdf which contained the details of the language.

During the implementation of the rules, I encountered some left recursion between 'expression' rule and the 'fragment_' rule. To solve this issue, I expanded the 'fragment_' with some rules from 'expression' until the error was gone.

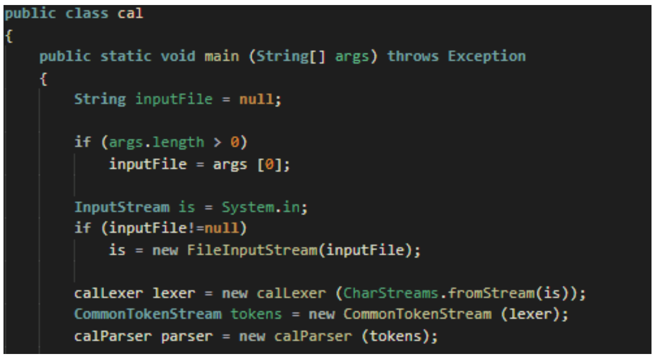


```
error(119): cal.g4::: The following sets of rules are mutually left-recursive [expression, fragment_]
```

Once I got the left-recursion sorted I started testing my ‘cal.g4’ file using ‘grun’ ’-gui’ command with different test files from the pdf to observes the parse tree I get.

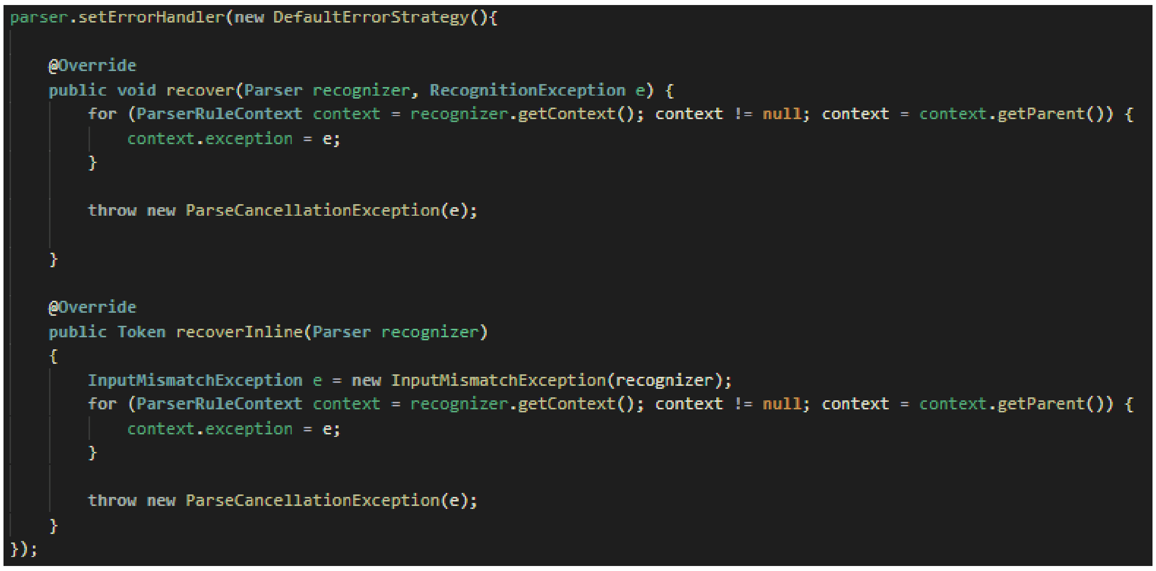
Implementation(cal.java):

This file contained will be used to do syntax analysis on the test files which were being provided to the grammar. I was asked to implement an error handling strategy which will either print out *‘file.cal parsed successfully’* or *‘file.cal has not parsed’* by catching some exception at runtime. The starting bit of the code is similar of what we did in labs.



```java
public class cal
{
    public static void main (String[] args) throws Exception
    {
        String inputFile = null;

        if (args.length > 0)
            inputFile = args [0];

        InputStream is = System.in;
        if (inputFile!=null)
            is = new FileInputStream(inputFile);

        calLexer lexer = new calLexer (CharStreams.fromStream(is));
        CommonTokenStream tokens = new CommonTokenStream (lexer);
        calParser parser = new calParser (tokens);
```

Now I had to implement my own error handler for which I started researching and found many different approaches of error handling in antlr4, some included listeners some made their own DefaultErrorStrategy by overriding this whole class but I decided to keeps things simple and use the methods use DefaultErrorStrategy but override any methods to my needs.



```java
parser.setErrorHandler(new DefaultErrorStrategy(){

    @Override
    public void recover(Parser recognizer, RecognitionException e) {
        for (ParserRuleContext context = recognizer.getContext(); context != null; context = context.getParent()) {
            context.exception = e;
        }

        throw new ParseCancellationException(e);

    }

    @Override
    public Token recoverInline(Parser recognizer)
    {
        InputMismatchException e = new InputMismatchException(recognizer);
        for (ParserRuleContext context = recognizer.getContext(); context != null; context = context.getParent()) {
            context.exception = e;
        }

        throw new ParseCancellationException(e);
    }
});
```

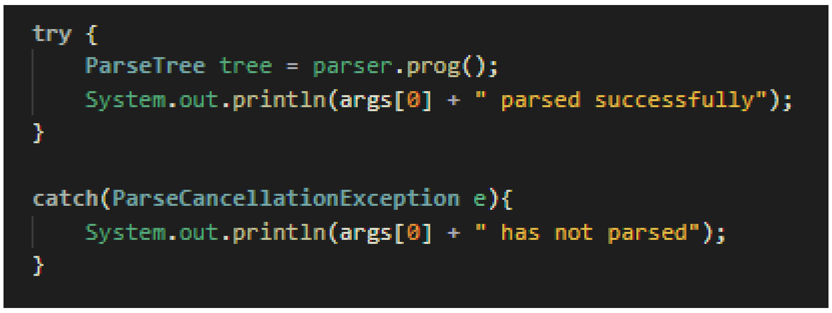
As I didn't know much about error handling in antlr4 in java, so I went on different websites to do my research such as;

- https://www.antlr.org/api/Java/org/antlr/v4/runtime/DefaultErrorStrategy.html

- Stackoverflow
    - https://stackoverflow.com/questions/30479367/antlr4-recover-from-error-and-continue-parsing-until-eof
    - https://stackoverflow.com/questions/46691795/consuming-error-tokens-in-antlr4
    - https://stackoverflow.com/questions/39533809/antlr4-how-to-detect-unrecognized-token-and-given-sentence-is-invalid
    - https://stackoverflow.com/questions/51683104/how-to-catch-minor-errors

These websites helped me gets bits and pieces of my error handling code together.

Now I just had to catch the exceptions I made in the overridden methods of the DefaultErrorStrategy and show appropriate messages.



```java
try {
    ParseTree tree = parser.prog();
    System.out.println(args[0] + " parsed successfully");
}

catch(ParseCancellationException e){
    System.out.println(args[0] + " has not parsed");
}
```