CA4003-Compiler Construction
Assignment 1

CAL(syntax and lexical analyser)

<u>Declaration of Own Work</u>

I declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged, and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

I/We have read and understood the referencing guidelines found at

http://www.dcu.ie/info/regulations/plagiarism.shtml
https://www4.dcu.ie/students/az/plagiarism and/or recommended in the assignment guidelines.

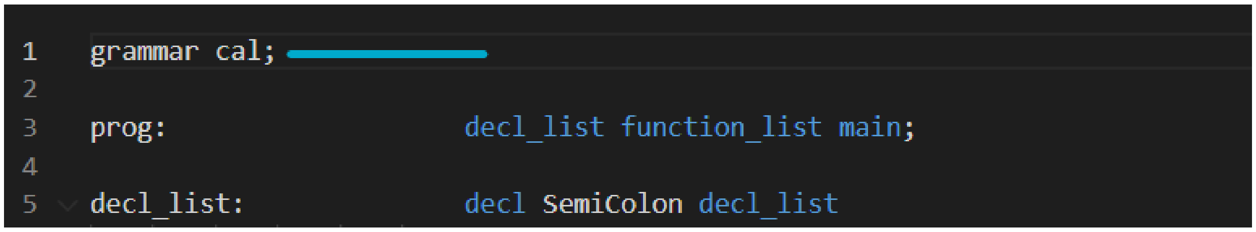
Name: Cian Crowley-Smith Date: 23/10/2021

Introduction

The assignment was to create a syntax and lexical analyser(parser) for a described programming language called CAL. In this document I will discuss my implementation for both and how I built the grammar for the programming language.

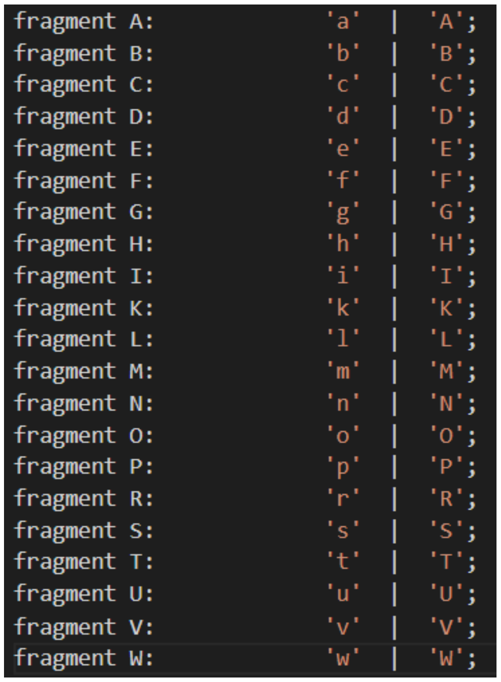
Implementation(cal.g4)

I created the grammar for the language by creating a file called “cal.g4” , the same name of the language. At the beginning of the file of specified the name of the grammar by writing the following lines

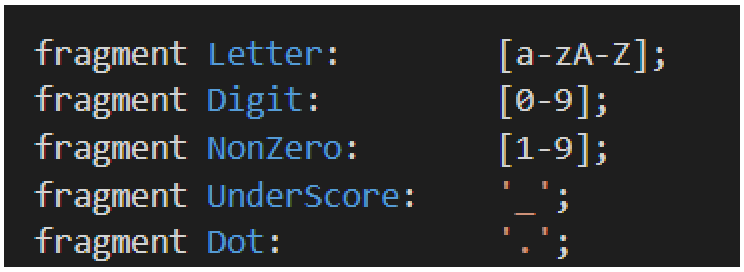


```
grammar cal;

prog:                   decl_list function_list main;

decl_list:              decl SemiColon decl_list
```

Cal is not case sensitive, so I had to specify this in the grammar by creating fragments for each letter in the alphabet then specifying another fragment with all letters of the alphabet to remove case sensitivity.



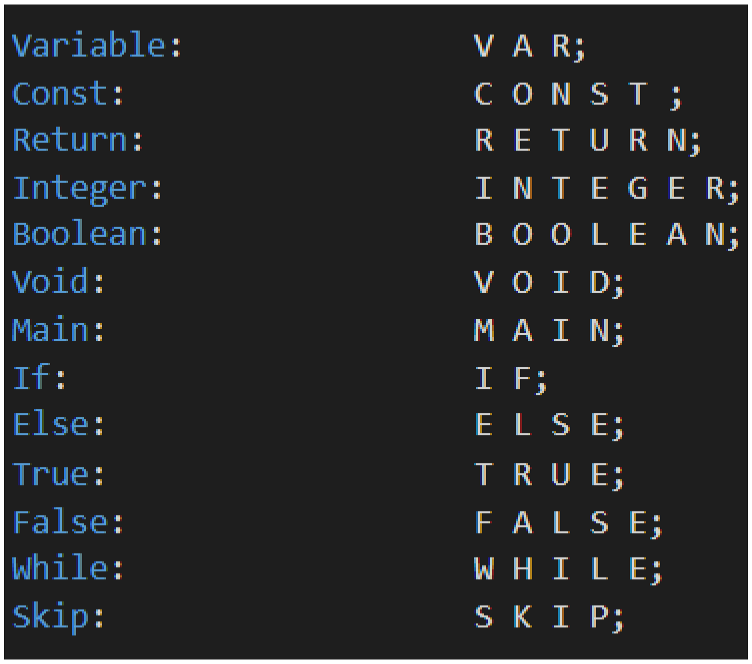
```
fragment A:        'a' | 'A';
fragment B:        'b' | 'B';
fragment C:        'c' | 'C';
fragment D:        'd' | 'D';
fragment E:        'e' | 'E';
fragment F:        'f' | 'F';
fragment G:        'g' | 'G';
fragment H:        'h' | 'H';
fragment I:        'i' | 'I';
fragment K:        'k' | 'K';
fragment L:        'l' | 'L';
fragment M:        'm' | 'M';
fragment N:        'n' | 'N';
fragment O:        'o' | 'O';
fragment P:        'p' | 'P';
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 Letter:        [a-zA-Z];
fragment Digit:         [0-9];
fragment NonZero:       [1-9];
fragment UnderScore:    '_';
fragment Dot:           '.';
```

I also Included “Dot”, “Underscore”, “Digit” and “NonZero” as fragments as they would be used in my parser rules.

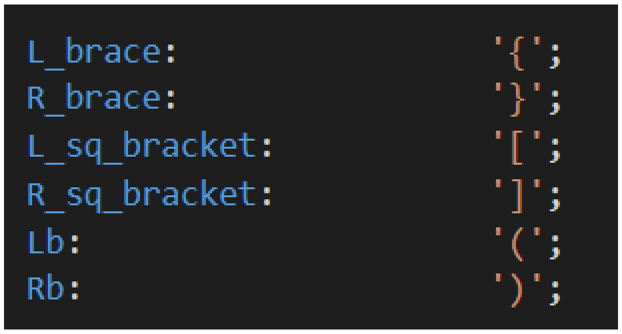
I then went ahead and implemented the reserved words that were specified in the language specification given in the assignment.



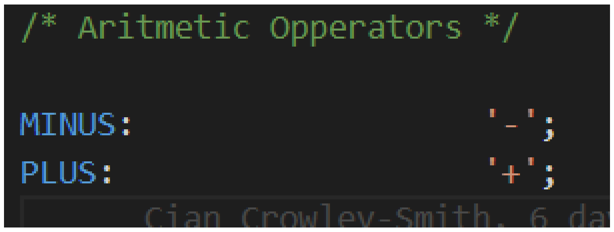
```
Variable:               V A R;
Const:                  C O N S T ;
Return:                 R E T U R N;
Integer:                I N T E G E R;
Boolean:                B O O L E A N;
Void:                   V O I D;
Main:                   M A I N;
If:                     I F;
Else:              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

In the specification it specifies the word “VAR” is a reserved word, I refer to it as Variable in this specification to provide clarity for what I am referring to in the grammar.

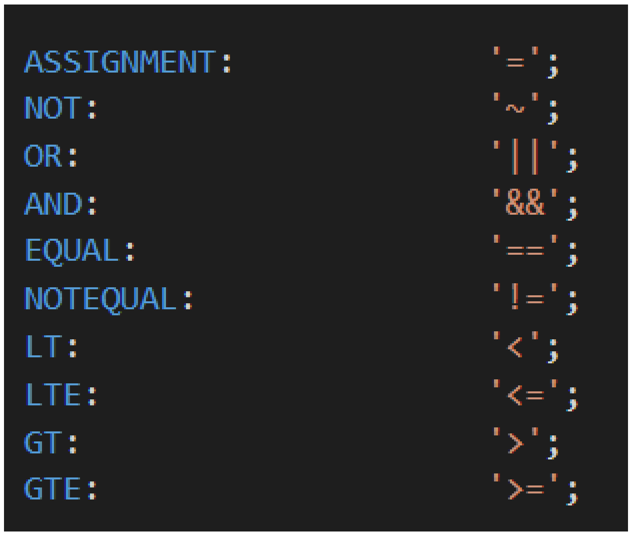
After specifying the reserved words I implemented all of the different types of operators in the language that would be used in the parser rules.



```
L_brace:                '{';
R_brace:                '}';
L_sq_bracket:           '[';
R_sq_bracket:           ']';
Lb:                     '(';
Rb:                     ')';
```

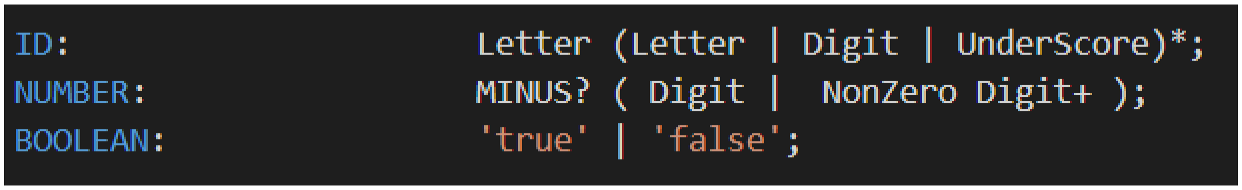


```
/* Aritmetic Opperators */

MINUS:                          '-';
PLUS:                           '+';
      Cian Crowley-Smith, 6 da
```



```
ASSIGNMENT:             '=';
NOT:                    '~';
OR:                     '||';
AND:                    '&&';
EQUAL:                  '==';
NOTEQUAL:               '!=';
LT:                     '<';
LTE:                    '<=';
GT:                     '>';
GTE:                    '>=';
```

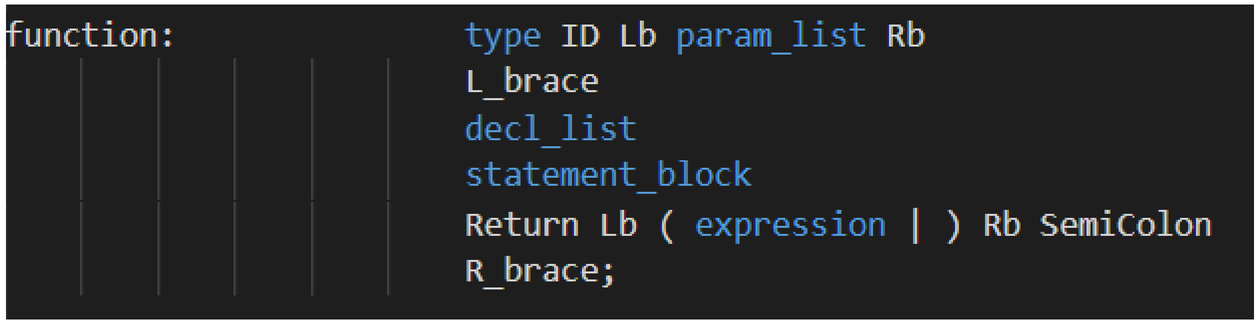
I also implemented the different types that would be available in the language such as Integers and Booleans as well as the Identifier type.



```
ID:                     Letter (Letter | Digit | UnderScore)*;
NUMBER:                 MINUS? ( Digit |  NonZero Digit+ );
BOOLEAN:                'true' | 'false';
```

These were combined with NonZero fragment and Minus fragment and also the underscore to give the desired effect of negative numbers and Identifiers that contain underscores are allowed.

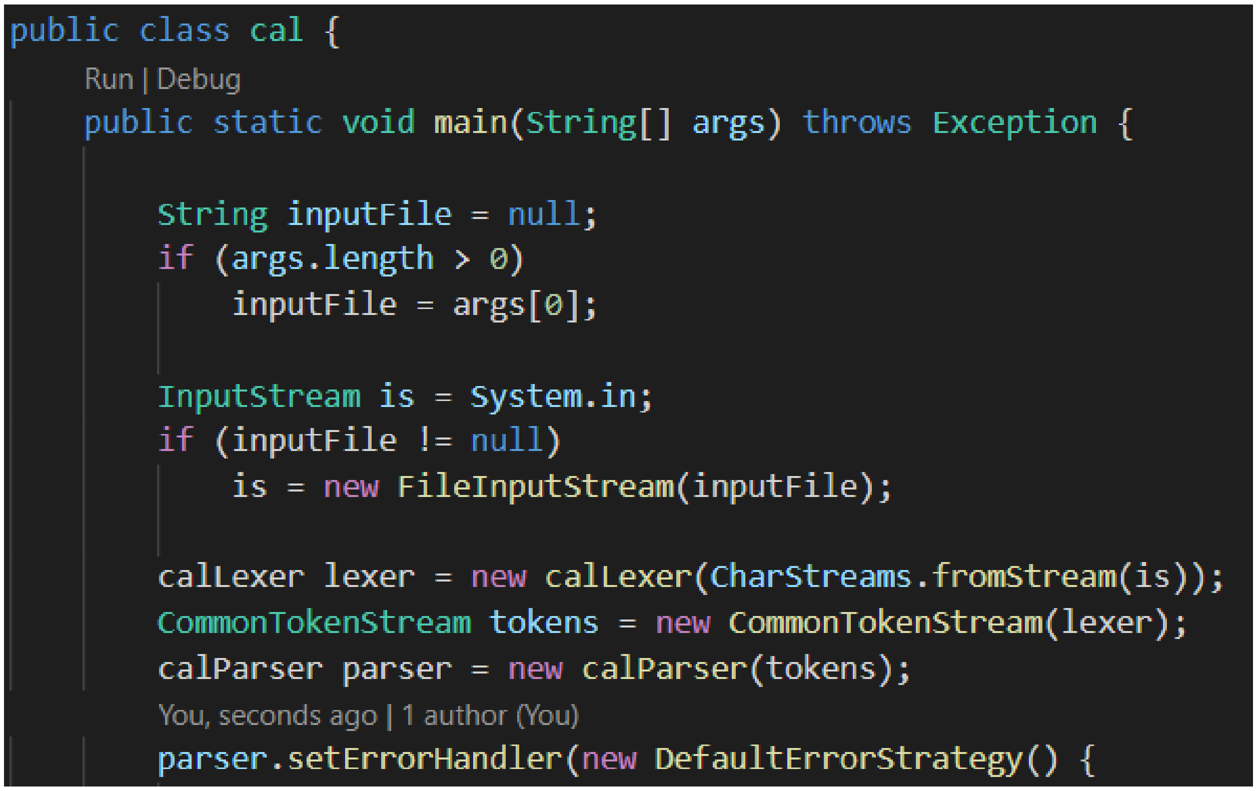
I then proceeded to complete my parsing rules which were just translating the given rules in the assignment specification into antlr form. I will attach an example of one below.



```
function:                   type ID Lb param_list Rb
                            L_brace
                            decl_list
                            statement_block
                            Return Lb ( expression | ) Rb SemiColon
                            R_brace;
```

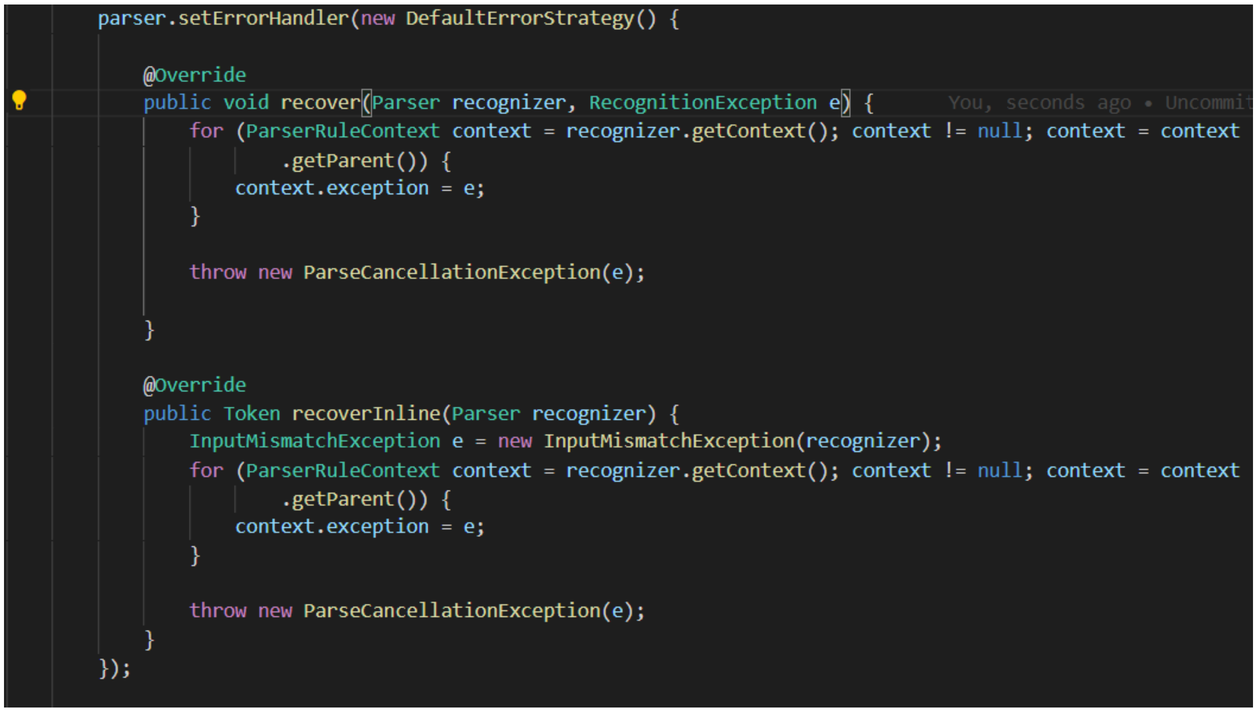
<u>Implementation(cal.java)</u>

After creating the grammar and running the antlr4 command with my grammar file to produce the required files for testing my grammar and to be able to visualise the abstract syntax tree with the grun command to verify that my grammar rules were correct, I began working on the semantic analyser which is the “cal.java” file. I was to correctly implement a parser that checked whether it was successfully parsed or not. I had to do some research on error handling with the antlr libraries that are provided by java and found the following solution.



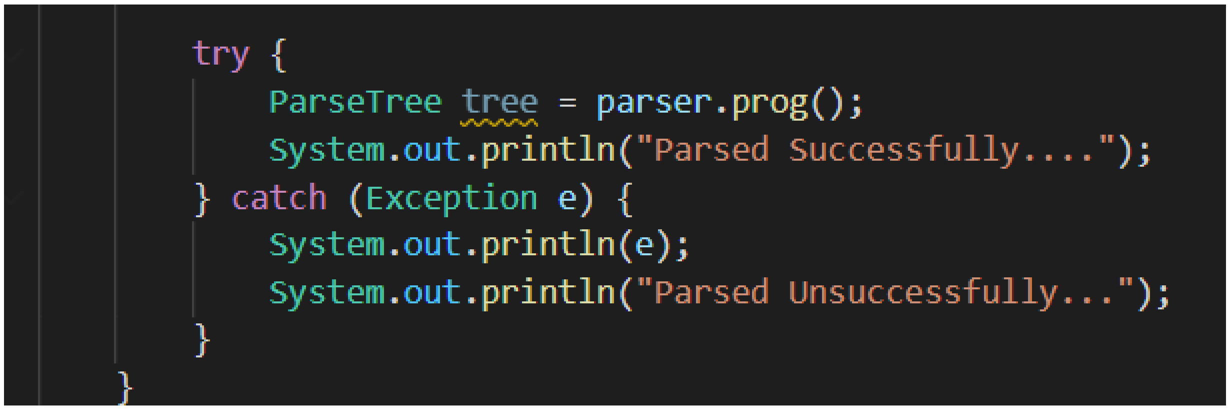
```
public class cal {
    Run | Debug
    public static void main(String[] args) throws Exception {

        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);
        You, seconds ago | 1 author (You)
        parser.setErrorHandler(new DefaultErrorStrategy() {
```

I took the main bulk of the parser from the video demonstrations as inspiration. After initializing the parser I had to add an error handler. I did some research and the easiest method was to implement my own error handler. I found some interesting and helpful websites to do with the antlr and error handling which I will attach at the bottom of this document



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

            @Override
            public void recover(Parser recognizer, RecognitionException e) {        You, seconds ago • Uncommit
                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 you can see from the above image I used the defaultErrorHandler and used “@overide” to alter the error handler to my needs. All of these methods are found in the antlr docs. After this my program runs a try catch block that runs the parser function and processes the given file.



```
        try {
            ParseTree tree = parser.prog();
            System.out.println("Parsed Successfully....");
        } catch (Exception e) {
            System.out.println(e);
            System.out.println("Parsed Unsuccessfully...");
        }
    }
```

<u>Resources:</u>

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

https://newbedev.com/handling-errors-in-antlr4

https://stackoverflow.com/questions/18132078/handling-errors-in-antlr4

https://tomassetti.me/antlr-mega-tutorial/