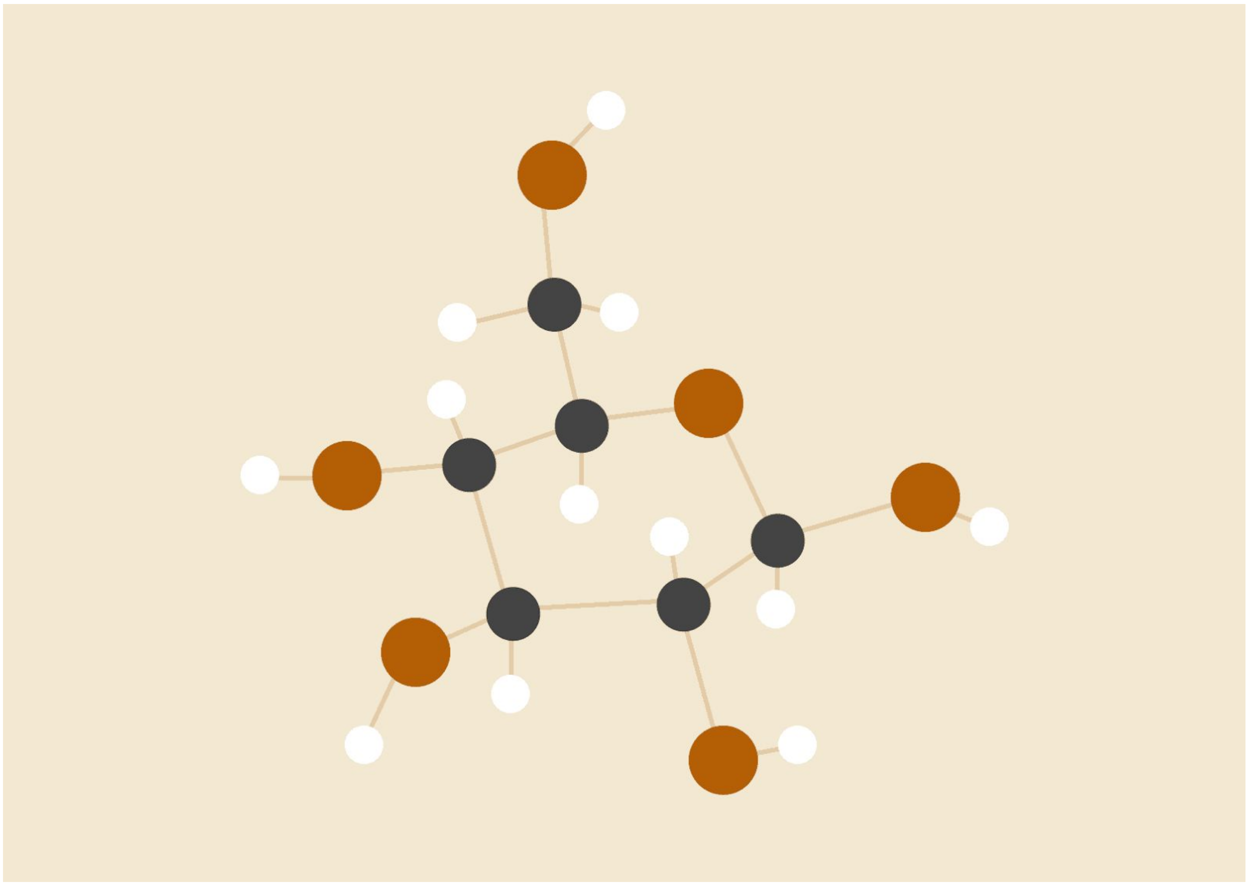
A Lexical and Syntax Analyser

CA4003 Assignment One



Niall Lyons - 13493628

12/11/18

CA4003 Compiler Construction

Declaration

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: Niall Lyons

Date: 12/11/18

INTRODUCTION

The aim of this assignment is to implement a lexical and syntax analyser using JavaCC for a simple language called CAL.

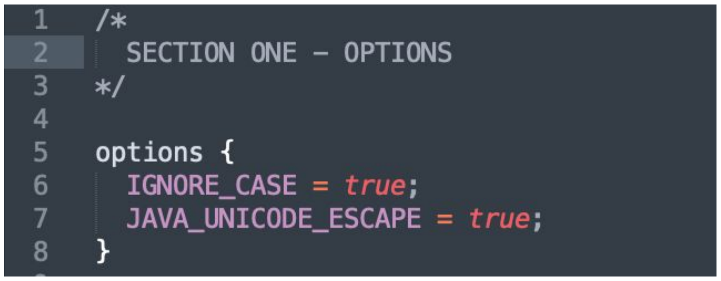
The details of the CAL language are given here.

I will be going through each of the four main components and briefly explaining how they were implemented. The four main sections of this report are:

- Options
- User Code
- Token Definitions
- Grammar and Production Rules

OPTIONS

The parser will have two options as seen below.

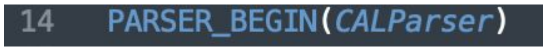


```
/*
  SECTION ONE - OPTIONS
*/

options {
  IGNORE_CASE = true;
  JAVA_UNICODE_ESCAPE = true;
}
```

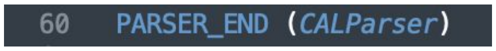
Option one, IGNORE_CASE is used as it is said in the details of CAL language that the language will not be case sensitive. Option two is the JAVA_UNICODE_ESCAPE, when set to true the generated parser uses an input stream object that processes Java Unicode escapes before sending characters to the token manager. By default, Java Unicode escapes are not processed.

USER CODE

This section holds the class definition of the parser, here we must enclose our code between the PARSER_BEGINS and PARSER_END keywords.

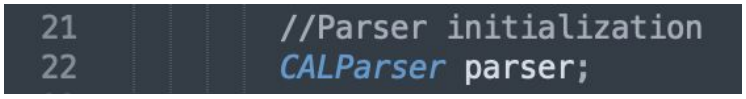


```
PARSER_BEGIN(CALParser)
```



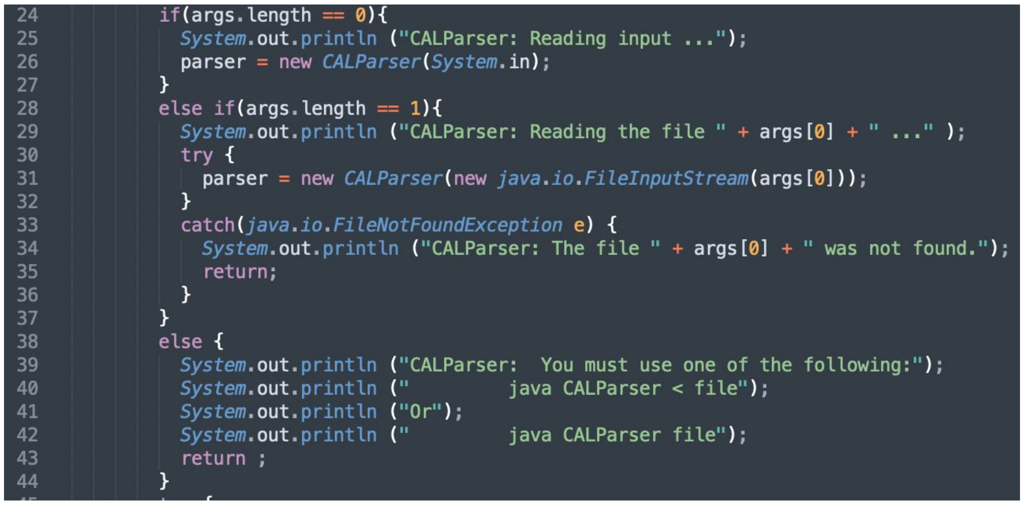
```
PARSER_END (CALParser)
```

The main method of the CALParser is pretty much the same as the SLPTokeniser parser code which is in the notes. I started off by initializing my parser by simply calling it parser as seen below.



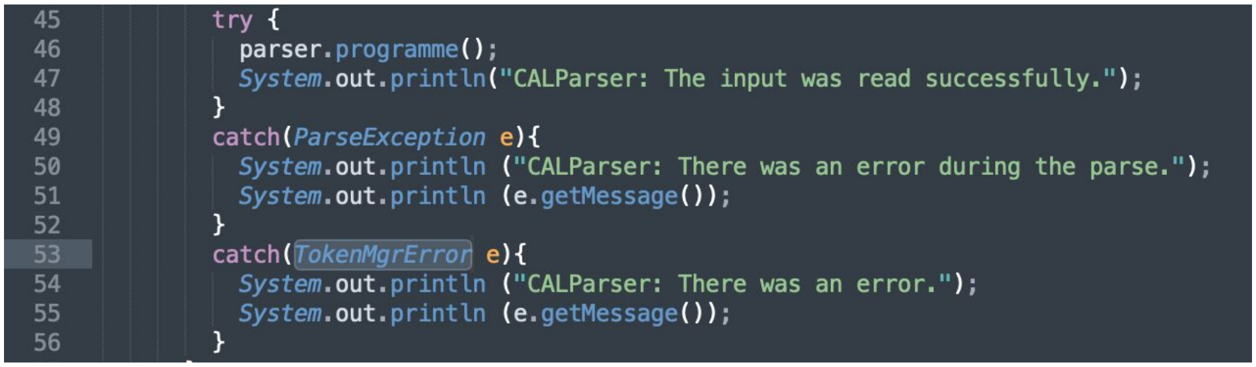
```
        //Parser initialization
        CALParser parser;
```

The code then goes on to initialise the parser that I have defined to read from the correct place. The parser will either begin to read input from standard input or from a file that is given as an argument from the command line. It also will also print out of the file name that was inputted cannot be found.



```
        if(args.length == 0){
          System.out.println ("CALParser: Reading input ...");
          parser = new CALParser(System.in);
        }
        else if(args.length == 1){
          System.out.println ("CALParser: Reading the file " + args[0] + " ..." );
          try {
            parser = new CALParser(new java.io.FileInputStream(args[0]));
          }
          catch(java.io.FileNotFoundException e) {
            System.out.println ("CALParser: The file " + args[0] + " was not found.");
            return;
          }
        }
        else {
          System.out.println ("CALParser:  You must use one of the following:");
          System.out.println ("         java CALParser < file");
          System.out.println ("Or");
          System.out.println ("         java CALParser file");
          return ;
        }
```

When the parser is initialized, it will then begin to try and parse either the input file or standard input from programme() which is defined below in the grammar section of the code. The parser will then go through the production rules and grammar to parse the file. This will result in either the file being parsed successfully, which will print a successful message or it will result in the file not being parsed successfully and will print the ParserException error message to the command line. I also use TokenMgrError here, to accommodate any errors that may have occurred in the token definitions.



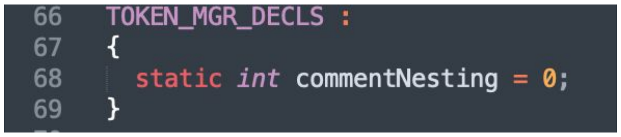
```
try {
  parser.programme();
  System.out.println("CALParser: The input was read successfully.");
}
catch(ParseException e){
  System.out.println ("CALParser: There was an error during the parse.");
  System.out.println (e.getMessage());
}
catch(TokenMgrError e){
  System.out.println ("CALParser: There was an error.");
  System.out.println (e.getMessage());
}
```

TOKEN DEFINITIONS

There are many different aspects of token definitions which must be thought about when developing a parser. I split this into four sections:

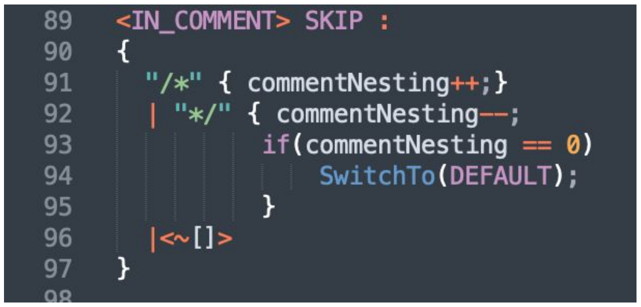
- **Comments**

  I first began by declaring the comment tokens which must be ignored. I started off by declaring a commentNesting variable which is defined in the JavaCC notes. This form of a comment begins with a " /* " and ends with a " */ ".



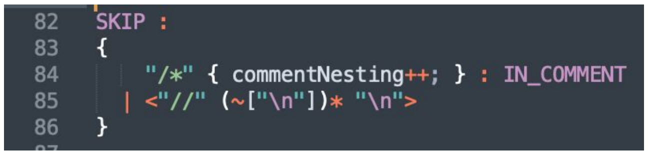
```
TOKEN_MGR_DECLS :
{
  static int commentNesting = 0;
}
```

  This comment nesting variable is incremented each time we see " /* " resulting in the lexical state being altered. We then move to a new state where the same variable is either decremented when we see " */ " and incremented when we see " /* ". After every decrementation, it must be checked if it is equal to zero, if the variable is zero we move back to the default lexical state. This is declared below:



```
<IN_COMMENT> SKIP :
{
  "/*" { commentNesting++;}
  | "*/" { commentNesting--;
        if(commentNesting == 0)
          SwitchTo(DEFAULT);
        }
  |<~[]>
}
```

I then had to take into consideration single line comments. This token is defined by:



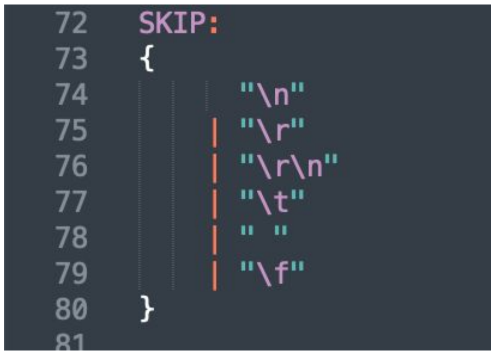
```
82  SKIP :
83  {
84      "/*" { commentNesting++; } : IN_COMMENT
85      | <"//" (~["\n"])* "\n">
86  }
```

Here we also take into consideration the commentNesting variable which we have defined before and increment it if we find “/* ” and use it in <IN_COMMENT> SKIP.

On line 85 above, <"//" (~["\n"])* "\n"> is defined. This will match any comments that begin with “ // ” and is the followed by anything other than a new line. This will go on until a new line character is seen.

- **Characters to be ignored**

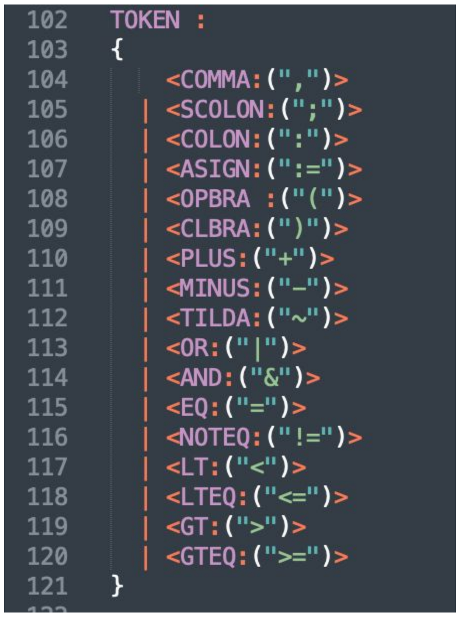
Another SKIP block is then created in order to ignore tabs, spaces and newlines as we dont want to read these in as tokens. This is shown below:



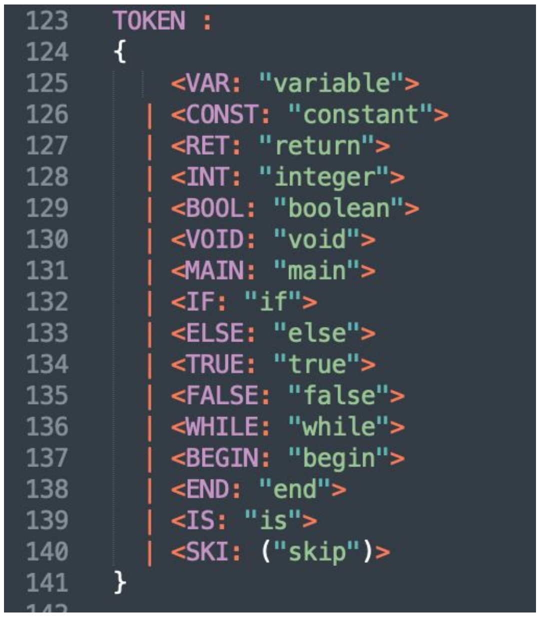
```
SKIP:
{
        "\n"
      | "\r"
      | "\r\n"
      | "\t"
      | " "
      | "\f"
}
```

- **Keyword and Operators**

To declare keywords and operators you first need to define it as a token. This is done by defining the token, and followed by the matching string. This can be all done in one token block, but for the ease of reading the code I decided to split them up into two as you can see below. All of the tokens that are seen below have been defined in the CAL language definition.



```
TOKEN :
{
    <COMMA:(",")>
  | <SCOLON:(";")>
  | <COLON:(":")>
  | <ASIGN:(":=")>
  | <OPBRA :("(")>
  | <CLBRA:(")")>
  | <PLUS:("+")>
  | <MINUS:("-")>
  | <TILDA:("~")>
  | <OR:("|")>
  | <AND:("&")>
  | <EQ:("=")>
  | <NOTEQ:("!=")>
  | <LT:("<")>
  | <LTEQ:("<=")>
  | <GT:(">")>
  | <GTEQ:(">=")>
}
```



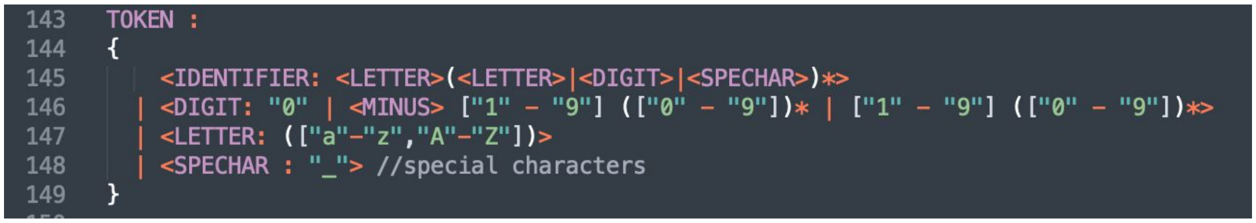
```
TOKEN :
{
    <VAR: "variable">
  | <CONST: "constant">
  | <RET: "return">
  | <INT: "integer">
  | <BOOL: "boolean">
  | <VOID: "void">
  | <MAIN: "main">
  | <IF: "if">
  | <ELSE: "else">
  | <TRUE: "true">
  | <FALSE: "false">
  | <WHILE: "while">
  | <BEGIN: "begin">
  | <END: "end">
  | <IS: "is">
  | <SKI: ("skip")>
}
```

As you can see above, I used <SKI> as it would not conflict with SKIP that I defined in above for the comments and skipping of characters.

- **Numbers and Identifiers**

In the language definition it reads " *Identifiers are represented by a string of letters, digits or underscore character ('_ ') beginning with a letter.* " and integers were defined as " *Integers are represented by a string of one or more digits ('0'-'9') and may start with a minus sign ('-'), e.g. 123, -456. Unless it is the number '0', numbers may not start with leading '0's* " .

Below is how I implemented this:

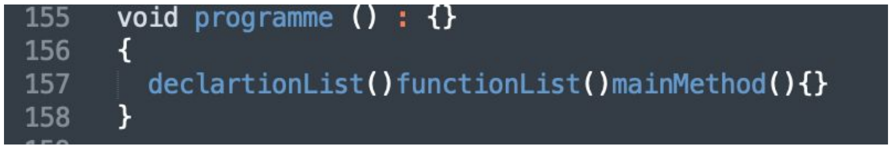


```
TOKEN :
{
    <IDENTIFIER: <LETTER>(<LETTER>|<DIGIT>|<SPECHAR>)*>
  | <DIGIT: "0" | <MINUS> ["1" - "9"] (["0" - "9"])* | ["1" - "9"] (["0" - "9"])*>
  | <LETTER: (["a"-"z","A"-"Z"])>
  | <SPECHAR : "_"> //special characters
}
```

As you can see I created a letter token which takes any letter from a-z and also in capital letters. I also created a special character to accommodate the identifier. I called all integers digits, here I accommodated all numbers represented by strings of one or more digits and can also start with a minus sign, unless it is 0. Here I reused the minus token I made before. I then created the identifier, this shows an identifier is a letter followed by zero or more letters, digits or special characters, this is because of the “ * ” which represents Kleene Closure.

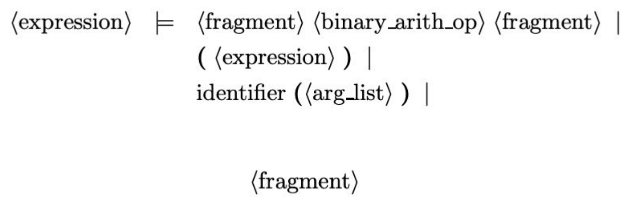
GRAMMAR AND PRODUCTION RULES

From the CAL language definition given I began to write out the grammar for my syntax analyser. I started by declaring my prog which the parser calls in the main method.



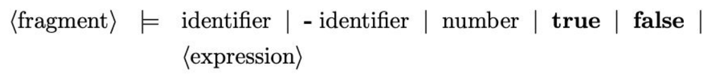
```
void programme () : {}
{
  declartionList()functionList()mainMethod(){}
}
```

I quickly noticed after writing these out that there was a few changes that I had to make. After compiling my JavaCC file I found that I had some left recursion errors. From the original grammar that was given there was left recursion present in expression and fragment non terminals. Originally fragment and expression looked like this:



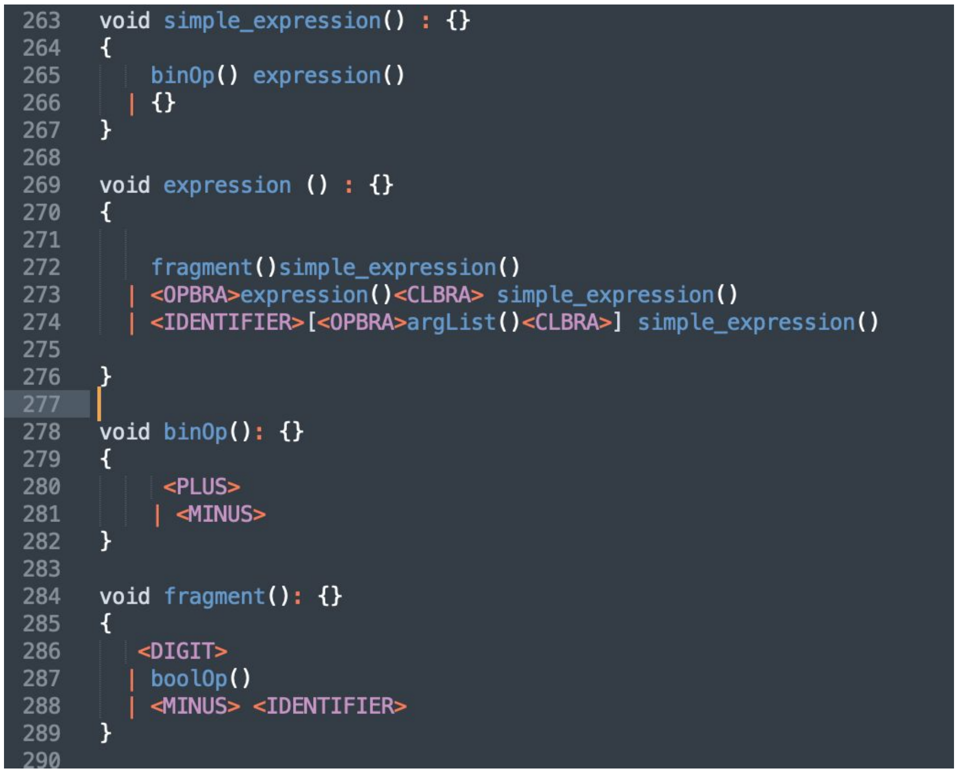
$\langle$expression$\rangle$ $\models$ $\langle$fragment$\rangle$ $\langle$binary_arith_op$\rangle$ $\langle$fragment$\rangle$ |
($\langle$expression$\rangle$) |
identifier ($\langle$arg_list$\rangle$) |

$\langle$fragment$\rangle$



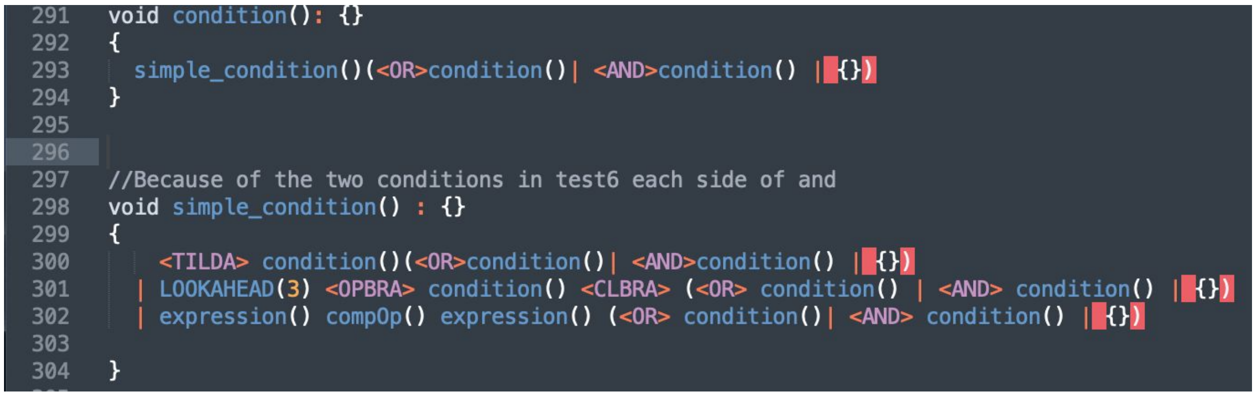
$\langle$fragment$\rangle$ $\models$ identifier | - identifier | number | **true** | **false** |
$\langle$expression$\rangle$

I began to develop a different approach to this problem by firstly creating a new simple_expression non terminal and adding this to the top line of my expression non terminal, meaning I would not be calling fragment twice. I then also took out the expression non terminal from fragment, and added in an optional binOp() expression() to the end of each production in expression() using my new simple_expression non terminal. The result of this can be seen in the code snippet below:



```
void simple_expression() : {}
{
    binOp() expression()
  | {}
}

void expression () : {}
{

    fragment()simple_expression()
  | <OPBRA>expression()<CLBRA> simple_expression()
  | <IDENTIFIER>[<OPBRA>argList()<CLBRA>] simple_expression()

}

void binOp(): {}
{
     <PLUS>
   | <MINUS>
}

void fragment(): {}
{
   <DIGIT>
  | boolOp()
  | <MINUS> <IDENTIFIER>
}
```

I then went on to fix the next left recursion error. I used a similar approach to fix the condition non terminal, by creating a new simple_condition non terminal and using the same method as above.

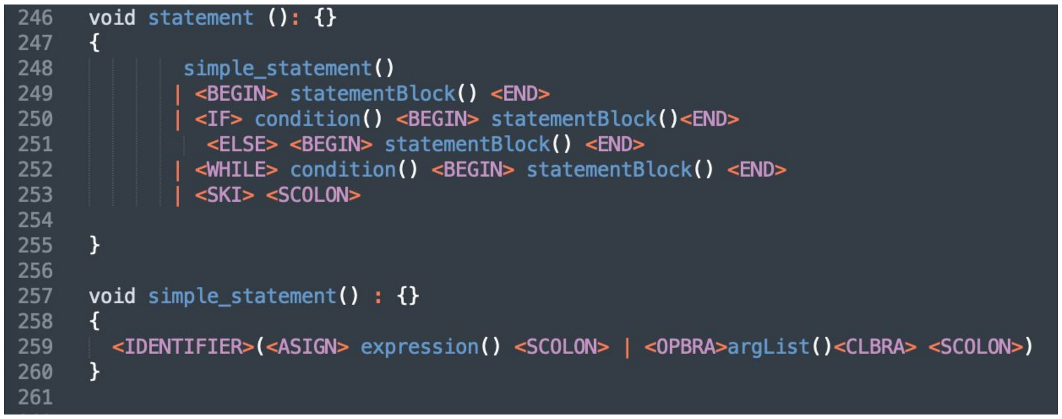


```
void condition(): {}
{
  simple_condition()(<OR>condition()| <AND>condition() |{})
}


//Because of the two conditions in test6 each side of and
void simple_condition() : {}
{
    <TILDA> condition()(<OR>condition()| <AND>condition() |{})
  | LOOKAHEAD(3) <OPBRA> condition() <CLBRA> (<OR> condition() | <AND> condition() |{})
  | expression() compOp() expression() (<OR> condition()| <AND> condition() |{})

}
```

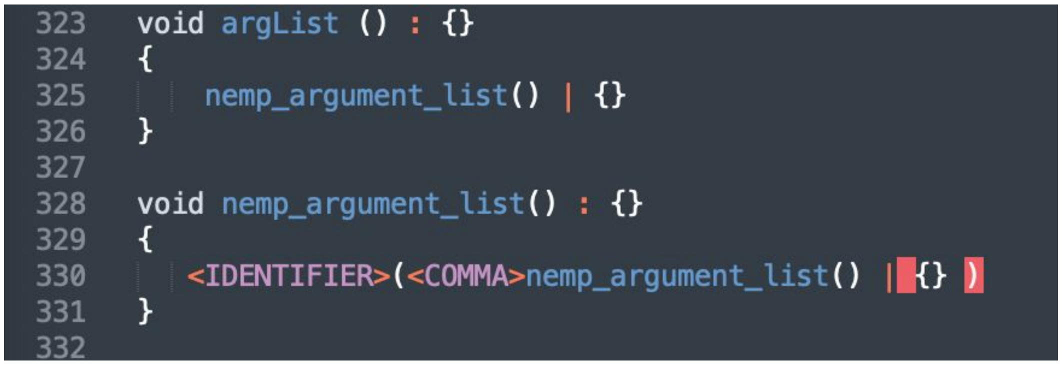
I also had to fix my statement non terminal as I had a choice conflict which original looked like this:

⟨statement⟩ ⊨ identifier := ⟨expression⟩ ; |
identifier (⟨arg_list⟩) ; |
begin ⟨statement_block⟩ **end** |
if ⟨condition⟩ **begin** ⟨statement_block⟩ **end**
else begin ⟨statement_block⟩ **end** |
while ⟨condition⟩ **begin** ⟨statement_block⟩ **end** |
skip ;

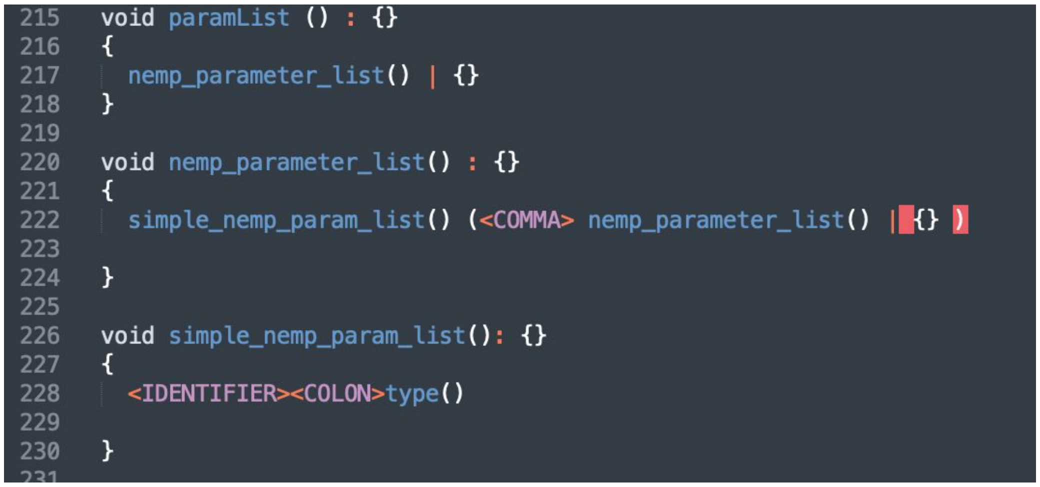
Here I made a new non terminal simple statement which satisfied statement. The resulting code looked like this:


```
void statement (): {}
{
        simple_statement()
      | <BEGIN> statementBlock() <END>
      | <IF> condition() <BEGIN> statementBlock()<END>
        <ELSE> <BEGIN> statementBlock() <END>
      | <WHILE> condition() <BEGIN> statementBlock() <END>
      | <SKI> <SCOLON>

}

void simple_statement() : {}
{
  <IDENTIFIER>(<ASIGN> expression() <SCOLON> | <OPBRA>argList()<CLBRA> <SCOLON>)
}
```

Where I also changed the non terminal nemp_argument_list() which uses argList(), this can be seen here:

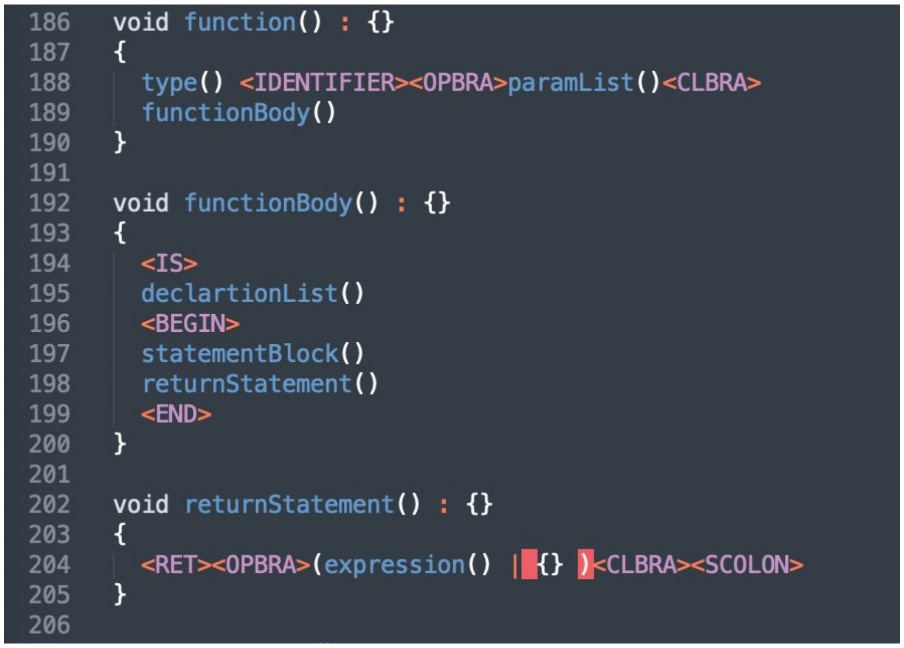

```
void argList () : {}
{
    nemp_argument_list() | {}
}

void nemp_argument_list() : {}
{
  <IDENTIFIER>(<COMMA>nemp_argument_list() | {} )
}
```

I also had to do this for nemp_parameter_list(), creating a new non terminal simple_nemp_parameter_list() as seen below:



```
void paramList () : {}
{
  nemp_parameter_list() | {}
}

void nemp_parameter_list() : {}
{
  simple_nemp_param_list() (<COMMA> nemp_parameter_list() | {} )

}

void simple_nemp_param_list(): {}
{
  <IDENTIFIER><COLON>type()

}
```

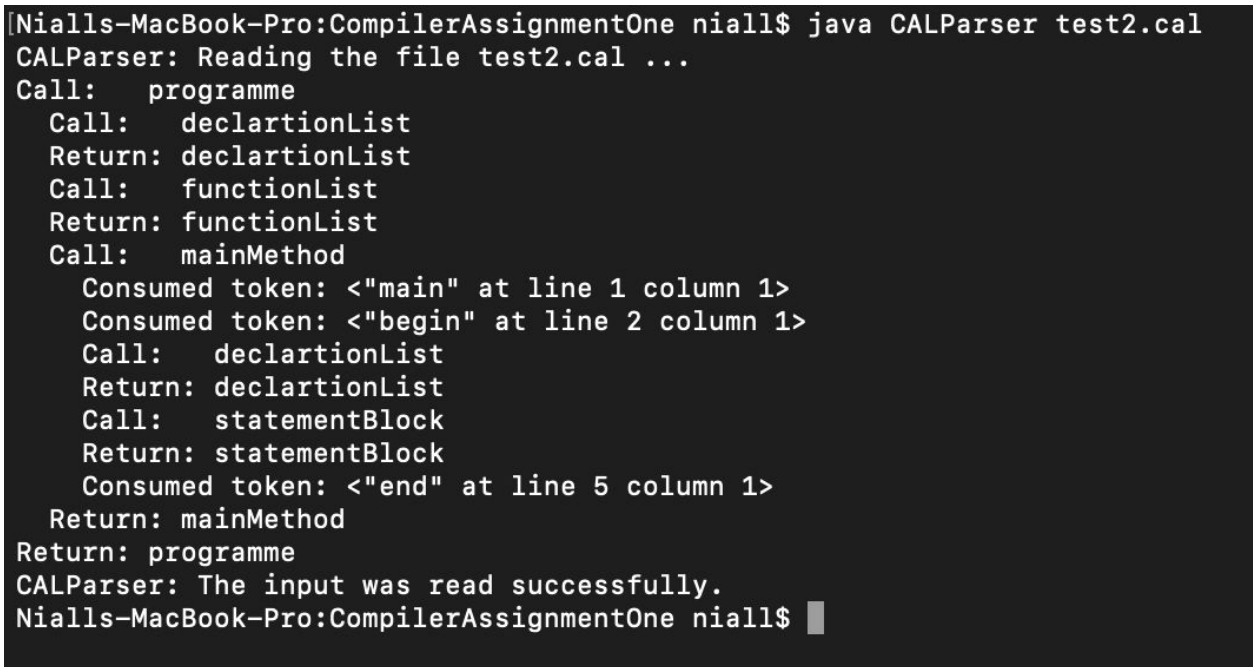
I also split up the function non terminal. The reason behind this was not only that I would understand what was happening more and also a better approach for ease of reading the code. I also thought that this would be better for assignment two. I did this by:



```
void function() : {}
{
  type() <IDENTIFIER><OPBRA>paramList()<CLBRA>
  functionBody()
}

void functionBody() : {}
{
  <IS>
  declartionList()
  <BEGIN>
  statementBlock()
  returnStatement()
  <END>
}

void returnStatement() : {}
{
  <RET><OPBRA>(expression() | {} )<CLBRA><SCOLON>
}
```

PARSING THE EXAMPLE CAL CODE

After I completed all the steps above I tried to compile the example CAL code tests that were given in the language definition file. The first two files parsed correctly, but then I had many unsuccessful parsing errors. Many were due to syntax errors in my code and spelling corrections. I made good use of the “ javacc -debug_parser ” option on the command line. This helped me pinpoint where I was going wrong and the slight changes I had to make. In most cases I had forgotten to add a specific rule. Running the debug parser looks like this after passing in a file.



```
[Nialls-MacBook-Pro:CompilerAssignmentOne niall$ java CALParser test2.cal
CALParser: Reading the file test2.cal ...
Call:   programme
  Call:   declartionList
  Return: declartionList
  Call:   functionList
  Return: functionList
  Call:   mainMethod
    Consumed token: <"main" at line 1 column 1>
    Consumed token: <"begin" at line 2 column 1>
    Call:   declartionList
    Return: declartionList
    Call:   statementBlock
    Return: statementBlock
    Consumed token: <"end" at line 5 column 1>
  Return: mainMethod
Return: programme
CALParser: The input was read successfully.
Nialls-MacBook-Pro:CompilerAssignmentOne niall$
```

HOW TO RUN MY PARSER

In order to compile this parser, you will need to run the following commands:

- Javacc assignment1.jj
- Javac *.java

To run my parser you will then need to run the following command:

- Java CALParser inputfile