CA4003-Assignment 1

Index

1. Introduction

The purpose of the assignment was to implement a lexical and syntax analyser for a simple language called CCAL. In order to achieve this we had to take the language specification which included the languages tokens and grammar, and write it in JavaCC with the goal of making it as efficient as possible.
In order to implement this four main sections were required to be done, these sections are options, the user code, token definitions and the grammar for the language.

2. Options

One of the requirements of the language specification was that the language need ***not*** be case sensitive. Looking through the JavaCC documentation I came across an option that would provide my lexical analyser with just that functionality. This option could be implemented at a local level meaning if a specific word need not be case sensitive or a global level which would make it applicative to all words. In this case the latter was chosen.

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

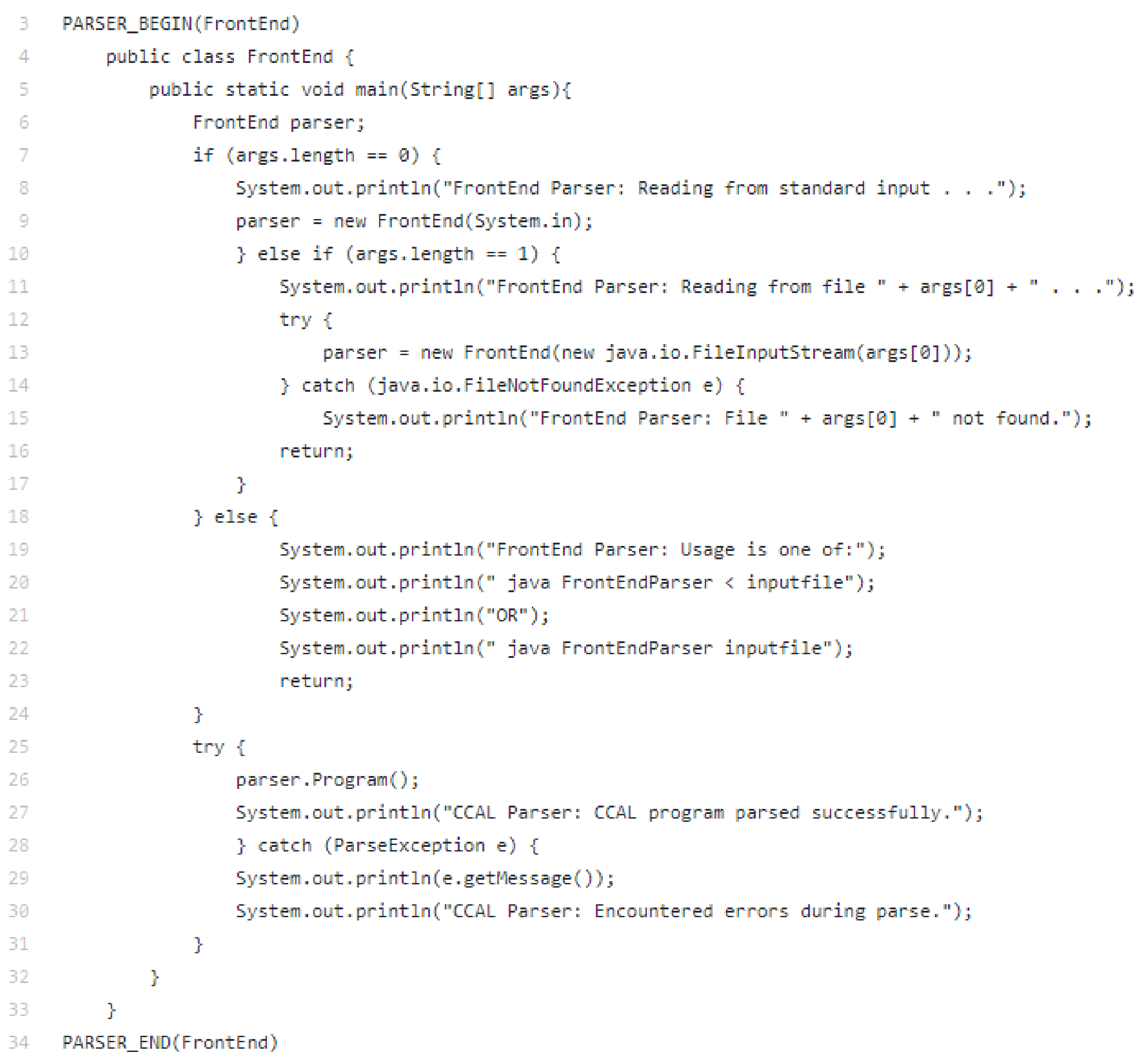
Figure 2.1

By adding this option our parser now accepts the following:

- MAIN
- mAiN
- main

3. User Code

The user code section primarily covers the parsers ability to be able to read in a particular file whether it is statically specified, read from standard input or passed as an argument vector and output whether or not the parser was able to parse the file successfully.



```
PARSER_BEGIN(FrontEnd)
    public class FrontEnd {
        public static void main(String[] args){
            FrontEnd parser;
            if (args.length == 0) {
                System.out.println("FrontEnd Parser: Reading from standard input . . .");
                parser = new FrontEnd(System.in);
                } else if (args.length == 1) {
                    System.out.println("FrontEnd Parser: Reading from file " + args[0] + " . . .");
                    try {
                        parser = new FrontEnd(new java.io.FileInputStream(args[0]));
                    } catch (java.io.FileNotFoundException e) {
                        System.out.println("FrontEnd Parser: File " + args[0] + " not found.");
                    return;
                }
            } else {
                    System.out.println("FrontEnd Parser: Usage is one of:");
                    System.out.println(" java FrontEndParser < inputfile");
                    System.out.println("OR");
                    System.out.println(" java FrontEndParser inputfile");
                    return;
            }
            try {
                parser.Program();
                System.out.println("CCAL Parser: CCAL program parsed successfully.");
                } catch (ParseException e) {
                System.out.println(e.getMessage());
                System.out.println("CCAL Parser: Encountered errors during parse.");
            }
        }
    }
PARSER_END(FrontEnd)
```

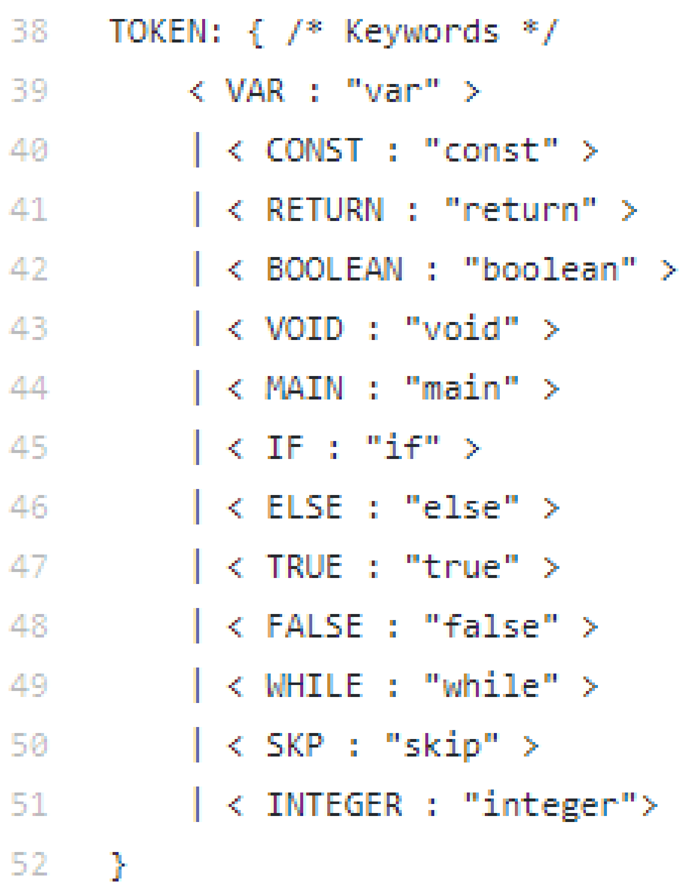
Figure 3.1

We can easily identify this section as it starts with *PARSER_BEGIN(FileName)* and finishes with *PARSER_END(FileName).*
The body of the user code is primarily default Java code however within the *try{...}* block we can see the *parser.Program()* method which is the important part of the user code section as without it the parser can't parse any files we pass to it. Even with the *parser.Program()* method present we must also include a catch for *ParseException* in the case that our parser cannot parse the file correctly due to either tokens that aren't recognized or grammar rules not being followed.

4. Token Definitions

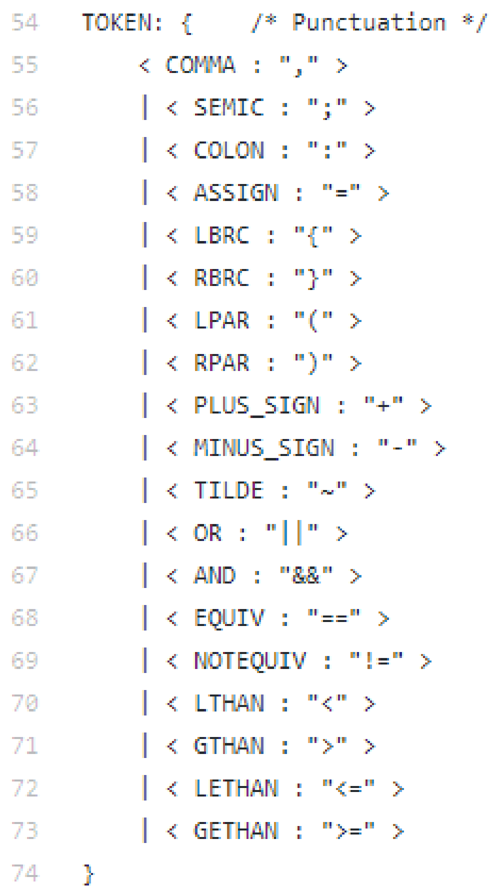
This section consists of the lexical analysis as we begin to reserve words, language specific symbols and data types, and what should be ignored or skipped by the parser, things that won't necessarily mean anything to the parser but have some relevance to whomever is reading the file.

4.1 Keywords & Punctuation



```
TOKEN: { /* Keywords */
    < VAR : "var" >
      | < CONST : "const" >
      | < RETURN : "return" >
      | < BOOLEAN : "boolean" >
      | < VOID : "void" >
      | < MAIN : "main" >
      | < IF : "if" >
      | < ELSE : "else" >
      | < TRUE : "true" >
      | < FALSE : "false" >
      | < WHILE : "while" >
      | < SKP : "skip" >
      | < INTEGER : "integer">
}
```

Figure 4.1.1

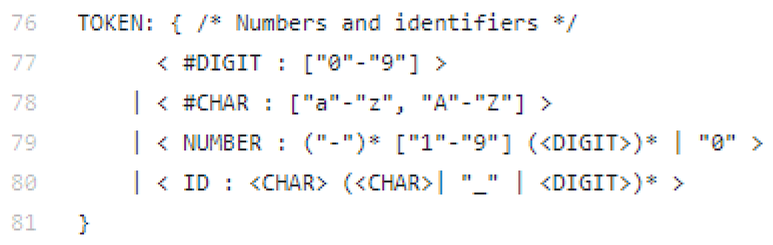


```
TOKEN: {    /* Punctuation */
    < COMMA : "," >
      | < SEMIC : ";" >
      | < COLON : ":" >
      | < ASSIGN : "=" >
      | < LBRC : "{" >
      | < RBRC : "}" >
      | < LPAR : "(" >
      | < RPAR : ")" >
      | < PLUS_SIGN : "+" >
      | < MINUS_SIGN : "-" >
      | < TILDE : "~" >
      | < OR : "||" >
      | < AND : "&&" >
      | < EQUIV : "==" >
      | < NOTEQUIV : "!=" >
      | < LTHAN : "<" >
      | < GTHAN : ">" >
      | < LETHAN : "<=" >
      | < GETHAN : ">=" >
}
```

Figure 4.1.2

In order for our parser to be able to lexically analyze anything we first of all tokenise all our keywords/reserved words of the language alongside the punctuation it will use.
In our case our keywords are clearly represented in *figure 4.1.1* whilst our punctuation is defined in *figure 4.1.2.* In this case I opted to divide each token section up into subsections as it promotes readability and in turn helped locate specific tokens when beginning to implement the syntax analyzer.

4.2 Numbers & Identifiers



```
TOKEN: { /* Numbers and identifiers */
      < #DIGIT : ["0"-"9"] >
    | < #CHAR : ["a"-"z", "A"-"Z"] >
    | < NUMBER : ("-")* ["1"-"9"] (<DIGIT>)* | "0" >
    | < ID : <CHAR> (<CHAR>| "_" | <DIGIT>)* >
}
```

Figure 4.2.1

When implementing integers and identifiers the following specification was given:

- Integers
  - string of one or more digits (‘0’-‘9’)
  - do not start with the digit ‘0’
  - may start with a minus sign (‘-’)
- Identifiers
  - string of
    - letters
    - digits
    - underscore character (‘ ’) beginning with a letter
  - cannot be reserved words

In order to implement this specification we made use of regular expressions (regex) and special tokens.

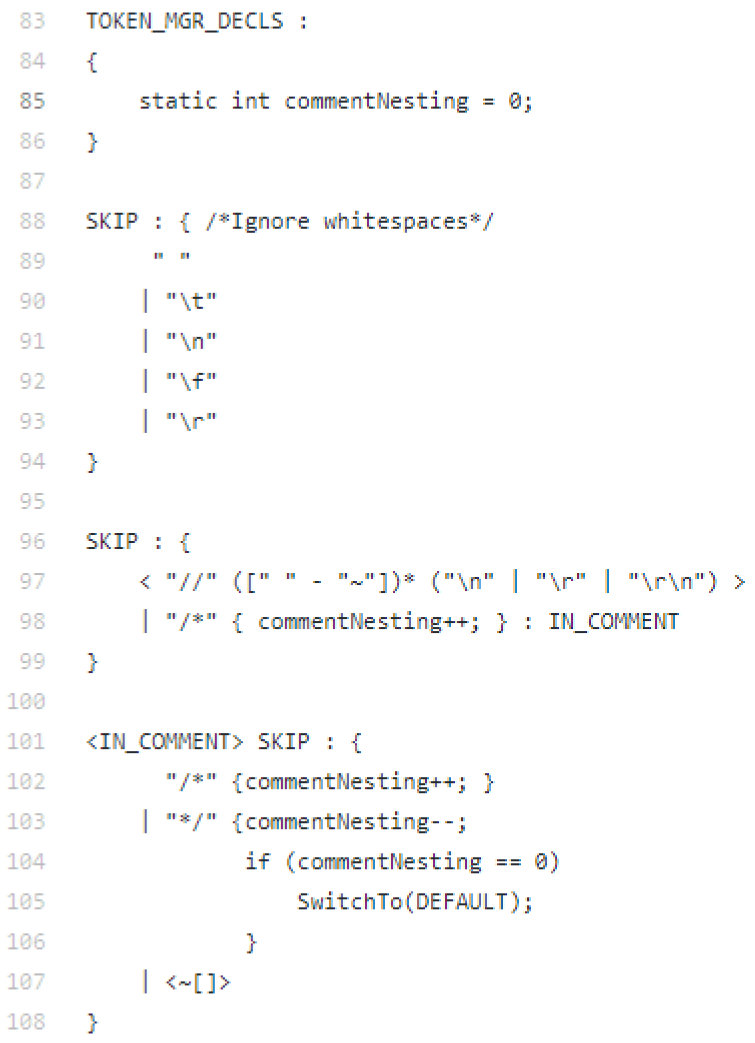
1. We divided the number specification into two parts, firstly we defined a special token (denoted by the hash symbol ‘#’) called *DIGIT* that represented a number in the range of 0 to 9
2. Then as per requirement an integer cannot begin with a '0' therefore we had to specify another range using regex which went from 1 to 9.
3. Finally the integer may start with a minus sign however it wasn’t specified that it had to be a single minus sign therefore we allowed there to be zero or many minus signs before an integer.
4. Following both those requirements we made use of our special token DIGIT and much like the minus sign we allowed it to appear zero or many times after our required 1 to 9 number.
5. If our integer did not match the first part of the regex then it would carry on past the pipe ‘|’ and check if the number was just a 0.

Anything outside of those 5 steps listed above is not considered a number.

The identifier follows the same idea as number by using the special token and more or less the same regex however the main difference here is that there is a sub section to the regex denoted by the expression enclosed in the parenthesis in *figure 4.2.1.*

1. An identifier must begin with a CHAR.
2. After that single CHAR it can be followed by either another CHAR, an underscore '_' or a DIGIT zero or many times.

4.3 SKIP



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

SKIP : { /*Ignore whitespaces*/
     " "
    | "\t"
    | "\n"
    | "\f"
    | "\r"
}

SKIP : {
    < "//" ([" " - "~"])* ("\n" | "\r" | "\r\n") >
    | "/*" { commentNesting++; } : IN_COMMENT
}

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

Figure 4.3.1

When parsing a file, all 'text' in that file is taken into account by that parser, this includes things we can and can't see but we don't want the parser to read therefore we need a way for the parser to omit checking this text. This is where the 'SKIP' section comes in handy, as we can see from *figure 4.3.1* we can get our parser to ignore white space, newline character, return character etc. We can also get it to

ignore blocks of text that match a particular regex, a perfect use case for this is comments.
The specification referred to two primary types of comments.

1. Single line comment denoted by "//".
2. Multi line comment denoted by "/*" and "*/".
    a. Which can be nested.

Single line comment is fairly straightforward regex, anything at all that follows a "//" up to either a newline character or return character or both is to be ignored.

The multi line comment follows the same idea as the single line comment but swaps the newline/return character for "*/" however in our case they must have the ability to be nested, in order to do this we make use of a local variable that if a "/*" is seen gets incremented and if a "*/" is seen is decremented, if this variable is 0 then we know that there are an equal amount of opening/closing brackets on both sides. The reason we must use a local variable is because regex doesn't have the ability to count.

5. Grammar

The secondary and more complex part of the assignment specification was to implement the context free grammar (CFG) given, in order to give our parser the ability to do syntax analyzing for our CCAL language. It is here in this section that we make heavy use of the previously defined tokens.

5.1 Context Free Grammar

The first step to implementing our grammar is to actually transcribe it across to our JavaCC program. Doing this is was a mechanical process because of the format that the grammar is presented in Backus-Naur form (BNF).
Using BNF we can easily identify:

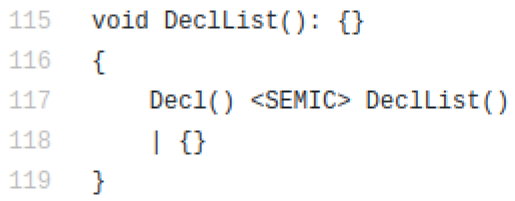
- terminals as the tokens we previously defined.
- non-terminals as the methods we will soon write.
- rules which is the root of a non-terminal and contains one or more productions.
- productions contain a sequence of terminals, non-terminals or both.

decl_list in BNF:

$$\langle \text{decl\_list} \rangle \models (\langle \text{decl} \rangle \ ; \ \langle \text{decl\_list} \rangle \mid \epsilon)$$

Figure 5.1.1

decl_list after being transcribed to JavaCC:



```
void DeclList(): {}
{
    Decl() <SEMIC> DeclList()
    | {}
}
```

Figure 5.1.2

> *Note: I've used "{}" instead of "?" to denote epsilon after transcribing as it promotes readability in my personal opinion.*

5.2 Left Factoring

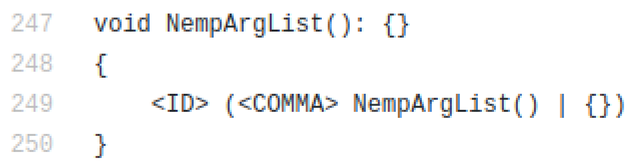
The main goal when implementing this parser was to make it as efficient as possible, this efficiency can be measured on its lookahead value, the higher the value the larger the table generated by JavaCC. Therefore the aim of this assignment was to keep the parser a LL(1) parser. However after transcribing the grammar to our JavaCC program there were occurrences where productions began with the same terminal or non-terminal which forces the parser to make a decision and if the wrong the decision was made it was forced to backtrack. We could fix this issue by increasing the lookahead value by N where N is the length of the common longest sub-string of terminals/non-terminals among a rules productions or we can apply the rule of left factoring and not increase the lookahead.

Example:

$$\langle \text{nemp\_arg\_list} \rangle \models \text{identifier} \mid \text{identifier} \ , \ \langle \text{nemp\_arg\_list} \rangle$$

Figure 5.2.1

In figure 5.2.1 we can see that 'identifier' is a common left factor among the productions. The left factoring rule generally implies that we create a new rule of prime and place the tails of both productions in it.



```
void NempArgList(): {}
{
    <ID> (<COMMA> NempArgList() | {})
}
```

Figure 5.2.2

However using a bit of common sense I found that figure 5.2.2 did the exact same whilst making the code a bit cleaner when left factoring the entire grammar.
During this assignment I found that applying left factoring first made things much easier with the exception of expression due to its effect when fixing the indirect left recursion which occurrs within the grammar which is covered down in section 5.4.

5.3 Direct Left Recursion

After left factoring if we attempted to compile our code we ran into the issue of left recursion.
This issue arose when we had a non-terminal where one of its productions began with that exact same non-terminal.
Much like left factoring the elimination of left recursion is mechanical by nature and therefore easy to apply.
When eliminating left recursion we can introduce a new prime of a non-terminal which expands the previously left recursive rule and apply the new prime of the rule to the end of each production and make it optional. By doing so we're simply restructuring our grammar in such a way that the tail end of the production that followed the left recursive production is now appended to the tail of each production and is optional, we can then remove the left recursive production resolving the issue while maintaining the grammar.

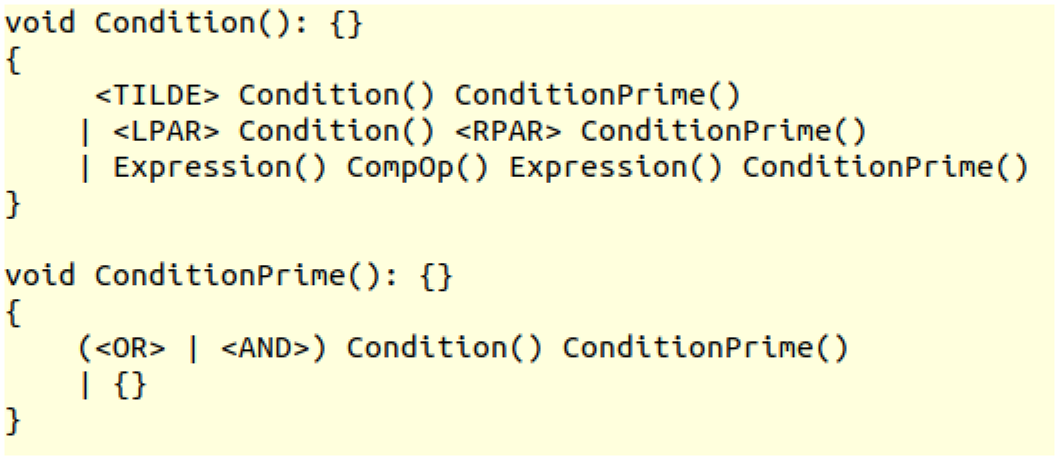
Example:

In the CFG given their was only one case of direct left recursion.

$$
\begin{array}{lcl}
\langle\text{condition}\rangle & \models & \sim\langle\text{condition}\rangle \;\mid \\
 & & (\;\langle\text{condition}\rangle\;)\;\mid \\
 & & \langle\text{expression}\rangle\;\langle\text{comp\_op}\rangle\;\langle\text{expression}\rangle\;\mid \\
 & & \langle\text{condition}\rangle(\;||\;\mid\;\&\&\;)\langle\text{condition}\rangle
\end{array}
$$

Figure 5.3.1

After applying the rule to eliminate direct left recursion we are left with:



```
void Condition(): {}
{
     <TILDE> Condition() ConditionPrime()
    | <LPAR> Condition() <RPAR> ConditionPrime()
    | Expression() CompOp() Expression() ConditionPrime()
}

void ConditionPrime(): {}
{
    (<OR> | <AND>) Condition() ConditionPrime()
    | {}
}
```

Figure 5.3.2

5.4 Indirect Left Recursion

Once all the direct left recursion was eliminated we were left with one final error, indirect left recursion. With indirect left recursion instead of the of a rule matching the left most non-terminal of one of its productions, one of the productions leads to a rule that then has a matching left most non terminal, the number of rules that it takes for this jump back up to the original rule can vary.

In the CFG that was given the rules that this was occurring to was *fragment* and *expression.*

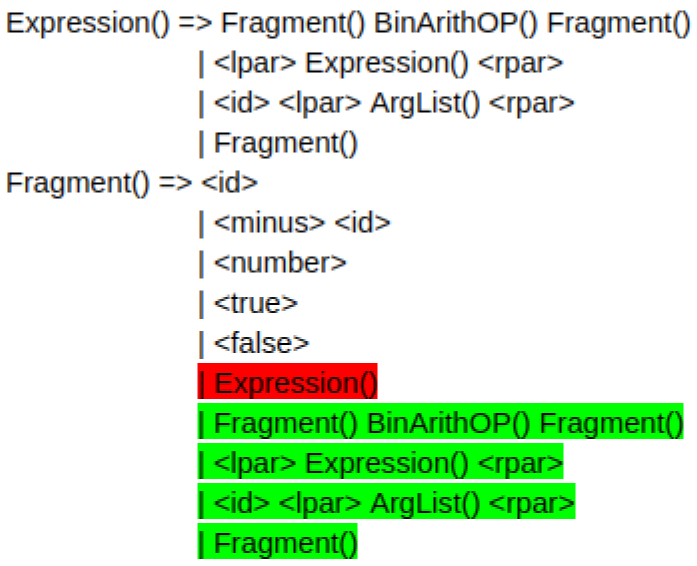
$$
\begin{array}{lcl}
\langle\text{expression}\rangle & \models & \langle\text{fragment}\rangle\;\langle\text{binary\_arith\_op}\rangle\;\langle\text{fragment}\rangle\;\mid \\
 & & (\;\langle\text{expression}\rangle\;)\;\mid \\
 & & \text{identifier}\;(\langle\text{arg\_list}\rangle\;)\;\mid \\
 & & \langle\text{fragment}\rangle
\end{array}
$$

Figure 5.4.1

$$\langle\text{fragment}\rangle \models \text{identifier} \mid \text{- identifier} \mid \text{number} \mid \textbf{true} \mid \textbf{false} \mid \langle\text{expression}\rangle$$

Figure 5.4.2

In figure 5.4.1 and 5.4.2 we can see that expression is calling fragment however fragment also calls expression creating an infinite loop. In order to eliminate this issue I took the contents of expression **BEFORE** I applied left factoring and replaced the contents of the expression production in the fragment rule which resulted in:

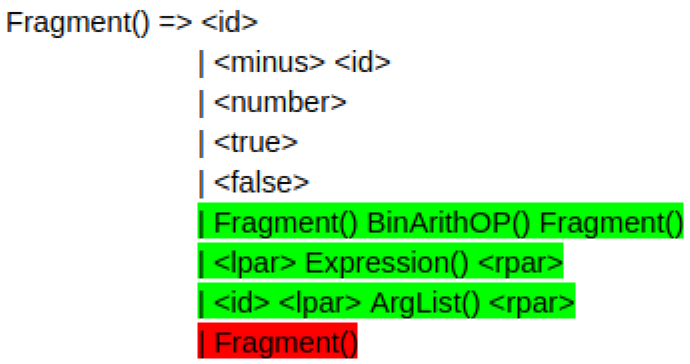


```
Expression() => Fragment() BinArithOP() Fragment()
              | <lpar> Expression() <rpar>
              | <id> <lpar> ArgList() <rpar>
              | Fragment()
Fragment() => <id>
              | <minus> <id>
              | <number>
              | <true>
              | <false>
              | Expression()
              | Fragment() BinArithOP() Fragment()
              | <lpar> Expression() <rpar>
              | <id> <lpar> ArgList() <rpar>
              | Fragment()
```

Figure 5.4.3

> *Red: What was removed*
> *Green: What was added*

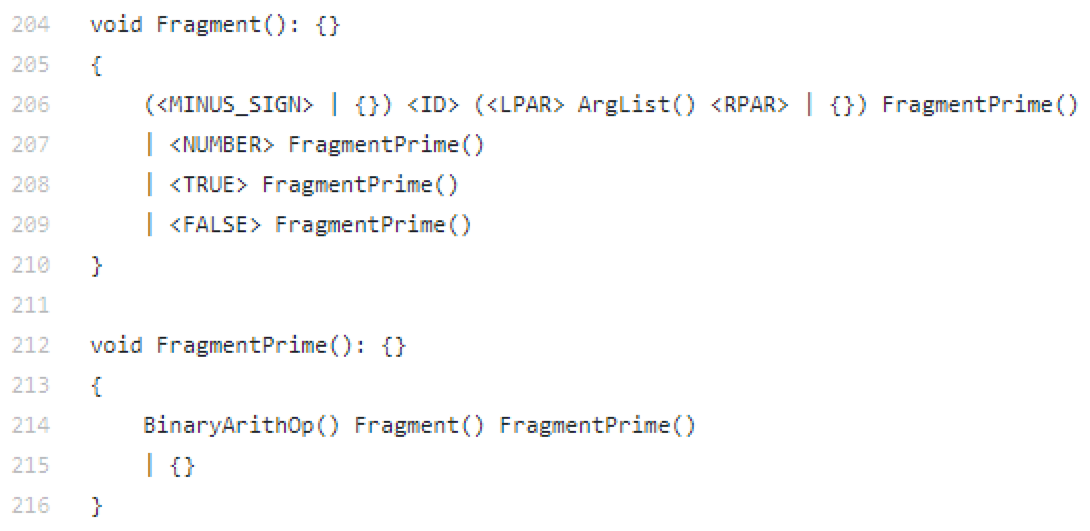
With the resulting fragment rule we now have an opportunity to apply left recursion however the idea of fragment going to fragment with nothing else in the production is nonsensical therefore we can simply remove the final production of fragment.



```
Fragment() => <id>
              | <minus> <id>
              | <number>
              | <true>
              | <false>
              | Fragment() BinArithOP() Fragment()
              | <lpar> Expression() <rpar>
              | <id> <lpar> ArgList() <rpar>
              | Fragment()
```

Figure 5.4.4

After removing the final production all that we are left with are direct left recursion, therefore we apply the rule to eliminate direct left recursion and we are left with the following.



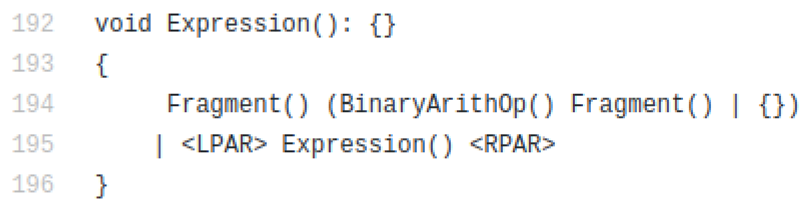
```
void Fragment(): {}
{
    (<MINUS_SIGN> | {}) <ID> (<LPAR> ArgList() <RPAR> | {}) FragmentPrime()
    | <NUMBER> FragmentPrime()
    | <TRUE> FragmentPrime()
    | <FALSE> FragmentPrime()
}

void FragmentPrime(): {}
{
    BinaryArithOp() Fragment() FragmentPrime()
    | {}
}
```

Figure 5.4.5

The reason we did not left factor first was because if we had our FragmentPrime rule would consist of two epsilons resulting in another left recursion error for our new prime rule.

Note in figure 5.4.5 for rule fragment the first production looks a lot more complicated than the CFG however it is simply aggregating the productions.
When the left recursion is eliminated we could return to expression and left factor it as seen below.



```
void Expression(): {}
{
     Fragment() (BinaryArithOp() Fragment() | {})
    | <LPAR> Expression() <RPAR>
}
```

Figure 5.4.6

5.5 Making the Parser LL1

After applying the rules described above we were able to compile and parse code samples however we were still being presented with warnings of common left factors even though we applied left factoring earlier.
However unlike the left factoring we previously talked about where a rule had common factors this left factoring branches across many rules that were linked

together indirectly sharing a common left factor. In order to fix these warnings we really had to delve into the grammar and understand the paths that it can take.

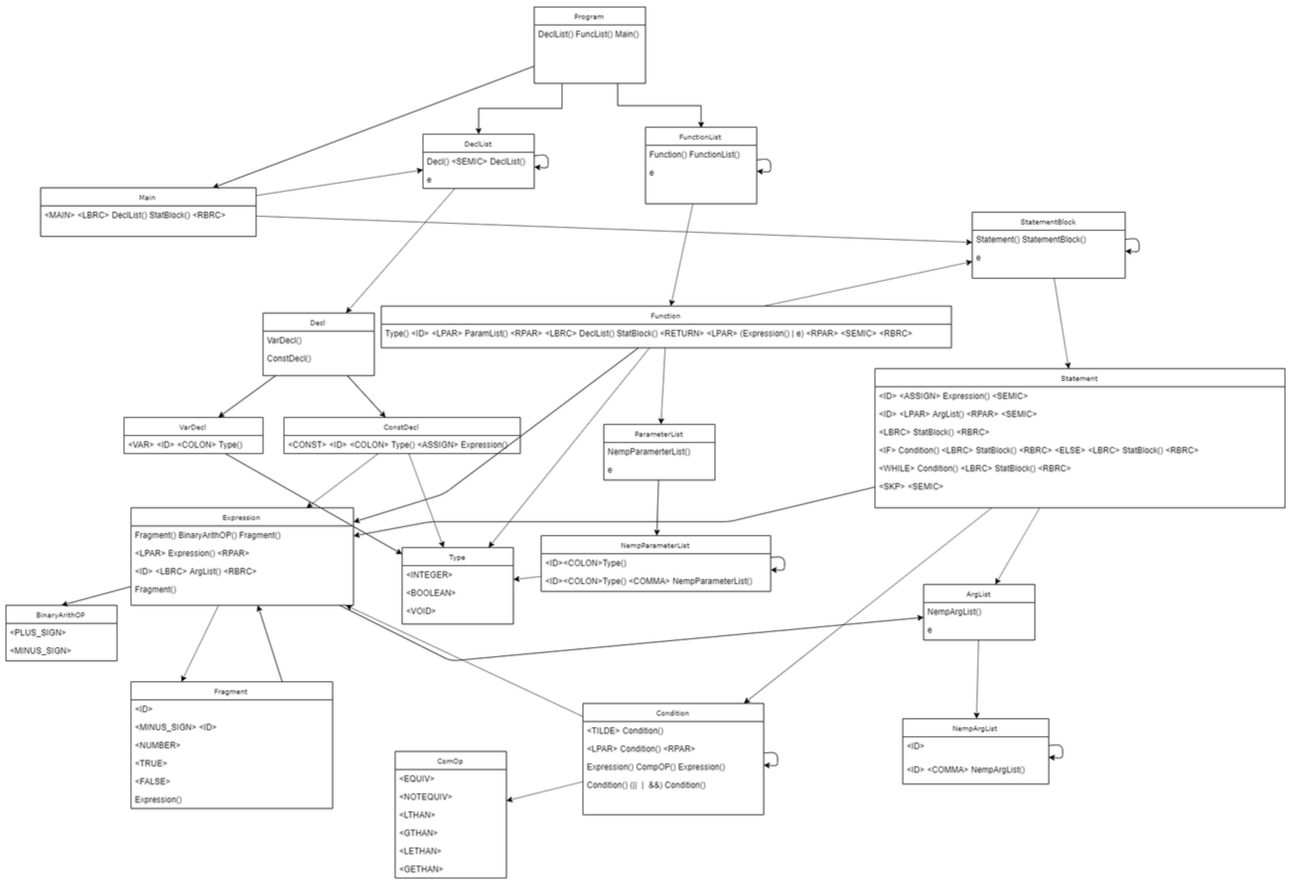
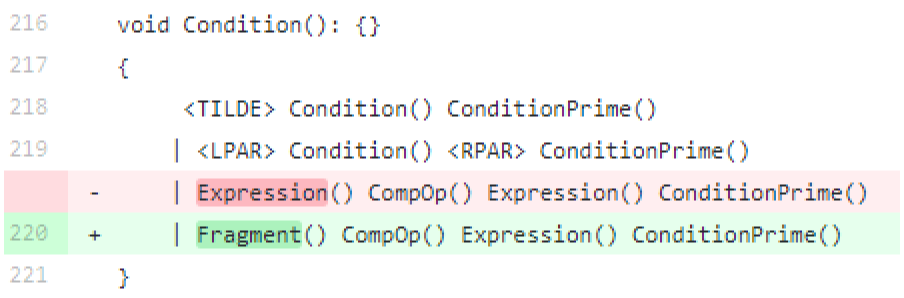

Program
DeclList() FuncList() Main()
DeclList
Decl() <SEMIC> DeclList()
e
FunctionList
Function() FunctionList()
e
Main
<MAIN> <LBRC> DeclList() StatBlock() <RBRC>
StatementBlock
Statement() StatementBlock()
e
Decl
VarDecl()
ConstDecl()
Function
Type() <ID> <LPAR> ParamList() <RPAR> <LBRC> DeclList() StatBlock() <RETURN> <LPAR> (Expression() | e) <RPAR> <SEMIC> <RBRC>
Statement
<ID> <ASSIGN> Expression() <SEMIC>
<ID> <LPAR> ArgList() <RPAR> <SEMIC>
<LBRC> StatBlock() <RBRC>
<IF> Condition() <LBRC> StatBlock() <RBRC> <ELSE> <LBRC> StatBlock() <RBRC>
<WHILE> Condition() <LBRC> StatBlock() <RBRC>
<SKP> <SEMIC>
VarDecl
<VAR> <ID> <COLON> Type()
ConstDecl
<CONST> <ID> <COLON> Type() <ASSIGN> Expression()
ParameterList
NempParamerterList()
e
Expression
Fragment() BinaryArithOP() Fragment()
<LPAR> Expression() <RPAR>
<ID> <LBRC> ArgList() <RBRC>
Fragment()
Type
<INTEGER>
<BOOLEAN>
<VOID>
NempParameterList
<ID><COLON>Type()
<ID><COLON>Type() <COMMA> NempParameterList()
ArgList
NempArgList()
e
BinaryArithOP
<PLUS_SIGN>
<MINUS_SIGN>
Fragment
<ID>
<MINUS_SIGN> <ID>
<NUMBER>
<TRUE>
<FALSE>
Expression()
Condition
<TILDE> Condition()
<LPAR> Condition() <RPAR>
Expression() CompOP() Expression()
Condition() (|| | &&) Condition()
NempArgList
<ID>
<ID> <COMMA> NempArgList()
CompOp
<EQUIV>
<NOTEQUIV>
<LTHAN>
<GTHAN>
<LETHAN>
<GETHAN>

Figure 5.5.1

> *Note: The grammar displayed above is the default grammar before any rules were applied.*

In the figure 5.5.1 we can see all the rules in our grammar with all their productions and all the paths they are able to take. When we compiled our code we found that the issue was stemming from Condition calling Expression as both rules had productions that had a common left factor of left parenthesis “(”.
In order to remove this issue we simply restructured our grammar to make Condition go to Fragment instead, and in turn Fragment can go to Expression. In order to achieve this, two changes were required to be made.

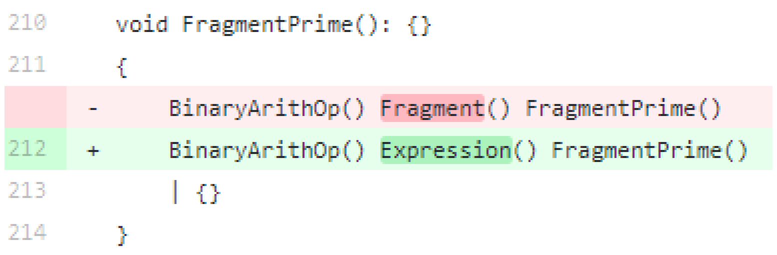
1. Condition going to Fragment



```
    void Condition(): {}
    {
        <TILDE> Condition() ConditionPrime()
       | <LPAR> Condition() <RPAR> ConditionPrime()
-      | Expression() CompOp() Expression() ConditionPrime()
+      | Fragment() CompOp() Expression() ConditionPrime()
    }
```

Figure 5.5.2

2. FragmentPrime going to Expression



```
    void FragmentPrime(): {}
    {
-       BinaryArithOp() Fragment() FragmentPrime()
+       BinaryArithOp() Expression() FragmentPrime()
        | {}
    }
```

Figure 5.5.3

The new paths created by these changes are shown in figure 5.5.4.

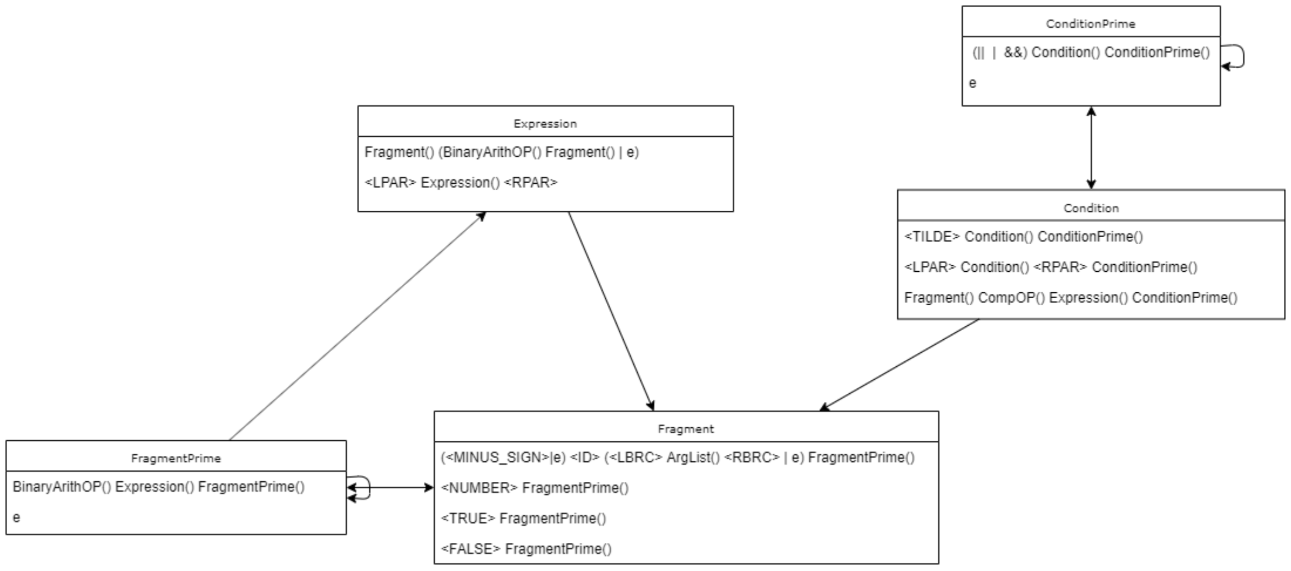
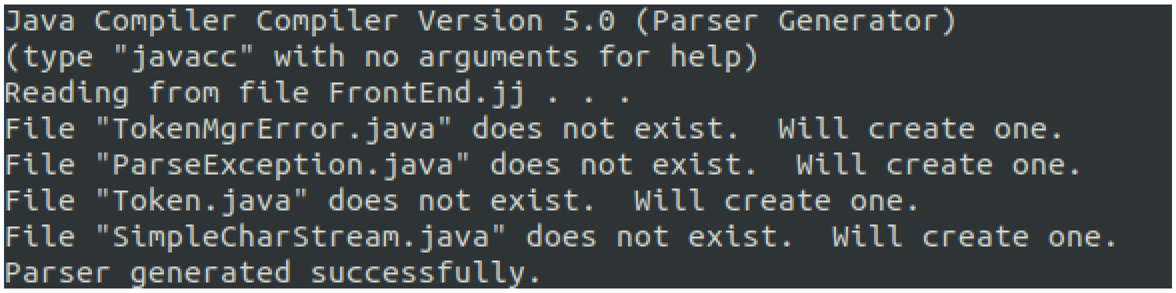

ConditionPrime
(|| | &&) Condition() ConditionPrime()
e
Expression
Fragment() (BinaryArithOP() Fragment() | e)
<LPAR> Expression() <RPAR>
Condition
<TILDE> Condition() ConditionPrime()
<LPAR> Condition() <RPAR> ConditionPrime()
Fragment() CompOP() Expression() ConditionPrime()
FragmentPrime
BinaryArithOP() Expression() FragmentPrime()
e
Fragment
(<MINUS_SIGN>|e) <ID> (<LBRC> ArgList() <RBRC> | e) FragmentPrime()
<NUMBER> FragmentPrime()
<TRUE> FragmentPrime()
<FALSE> FragmentPrime()

Figure 5.5.4

As we can see in figure 5.5.4 all that was done was adding an additional step into the path much like what we did when removing left recursion and introudcing a new rule of prime. By splitting out our path we remove ambiguity for the parser.

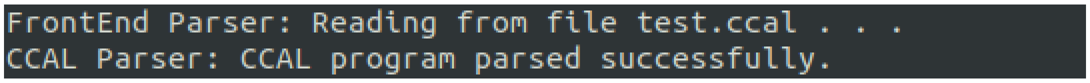
After these changes we compiled our parser using JavaCC and got the following output.



```
Java Compiler Compiler Version 5.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file FrontEnd.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

Figure 5.5.5

Once the parser was generated we then compiled the generated java files and ran our test.



```
FrontEnd Parser: Reading from file test.ccal . . .
CCAL Parser: CCAL program parsed successfully.
```

5.5.6

6. Conclusion

In conclusion the majority of the grammar was straightforward to implement since all that was required was to transcribe from the CFG given into our JavaCC program whilst occassionally applying some rules however when we got to the rules of Expression, Fragment and Condition thats when it got pretty tricky and a lot more foresight and understanding was required of both the grammar itself and the rules used to simplify it.