CA4003 Compiler Construction Assignment 1 Report

Student Name: Jake Grogan

Student Number: 16456346

Module: CA4003 Compiler Construction

Title: Assignment 1 - A Lexical and Syntax Analyser for the CCAL Language

Module Coordinator: David Sinclair

Table of Content

- Lexical Analysis
  - Overview
  - Implementation
- Syntax Analysis
  - Overview
  - Implementation
  - Testing
- Conclusion
- References

Lexical Analysis

Overview

Lexical analysis is the first phase of the compiler. It converts the input program into a sequence of tokens using a scanner.

The scanner must recognise various parts of the language such as whitespace, comments, keywords and operators.

The lexical analyser should do the following:

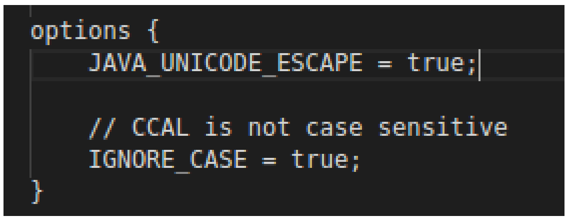
- Tokenize the input program into valid tokens
- Remove whitespace, tab, newline characters etc.

- Remove comments

Imlementation

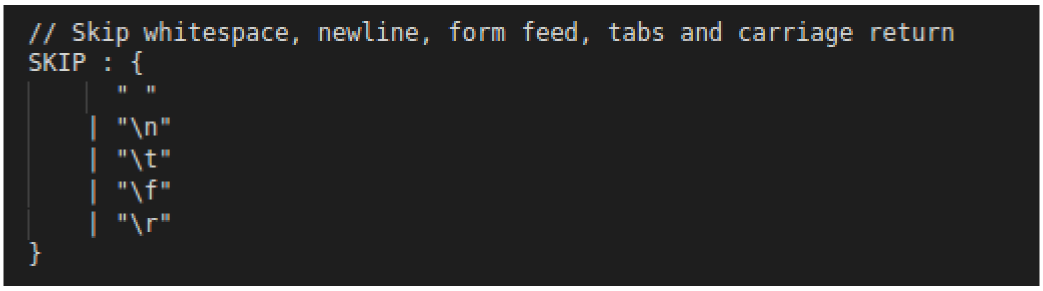
This section corresponds to section 3 of the source file (CCALParser.jj) SECTION 3: TOKEN DEFINITIONS

The CCAL language is not case sensitive, to ensure this is the case, I set the option `IGNORE_CASE` to true in the options section (SECTION 1).



```
options {
    JAVA_UNICODE_ESCAPE = true;

    // CCAL is not case sensitive
    IGNORE_CASE = true;
}
```

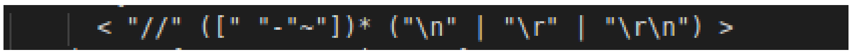
Next, I ensured the parser would ignore whitespace, tabs, newline, formfeed and carriage return characters. To do this is I created a SKIP pattern.



```
// Skip whitespace, newline, form feed, tabs and carriage return
SKIP : {
        " "
    | "\n"
    | "\t"
    | "\f"
    | "\r"
}
```

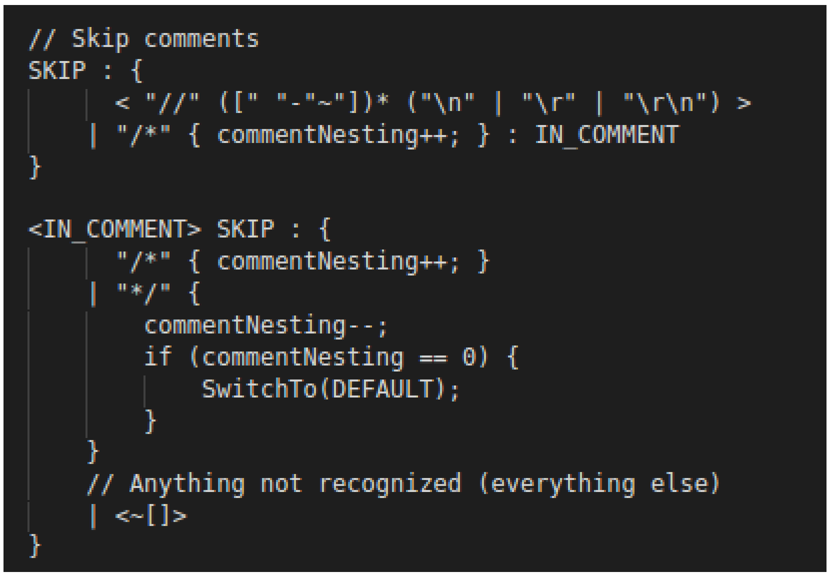
Next, I added a second SKIP pattern to ignore comments. There were two cases here, single line and multiline comments. In the case of multiline comments, I also had to ensure to ignore nested comments.

To ignore single line comments I implemented the following regular expression in a SKIP production:



```
    < "//" ([" "-"~"])* ("\n" | "\r" | "\r\n") >
```

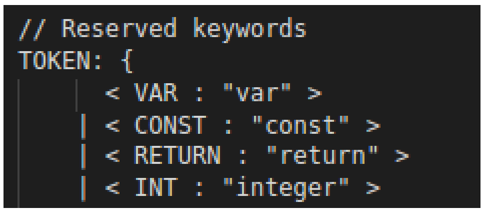
Multi-line comments are opened using the `/*` token and are closed using the `*/` token. I had to keep track of the number of opening comment tokens. If I did not do this, not all the content within the comments would be ignored. I did this using the following code inside a SKIP production:



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

<IN_COMMENT> SKIP : {
      "/*" { commentNesting++; }
    | "*/" {
        commentNesting--;
        if (commentNesting == 0) {
            SwitchTo(DEFAULT);
        }
    }
    // Anything not recognized (everything else)
    | <~[]>
}
```

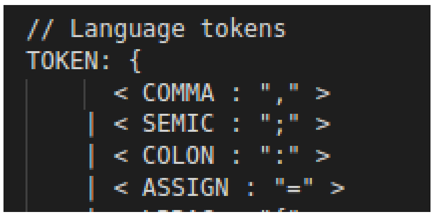
Where commentNesting keeps track of the 'depth' of the nested comments

Next I defined the reserved keywords of the language. This was done using a TOKEN production. A snippet is found below:



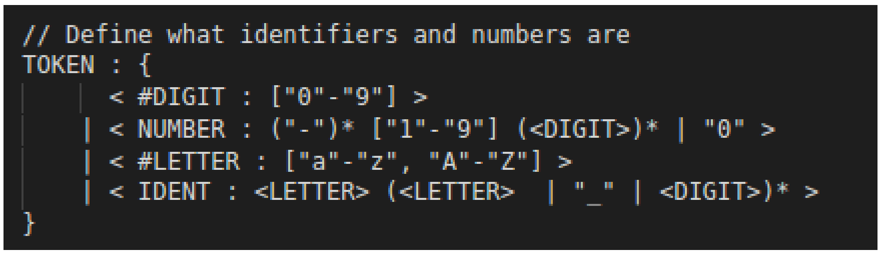
```
// Reserved keywords
TOKEN: {
      < VAR : "var" >
    | < CONST : "const" >
    | < RETURN : "return" >
    | < INT : "integer" >
```

Next I defined the tokens of the CCAL language using a second TOKEN production.



```
// Language tokens
TOKEN: {
      < COMMA : "," >
    | < SEMIC : ";" >
    | < COLON : ":" >
    | < ASSIGN : "=" >
```

My next task for implementing the lexical analyser was to define what numbers and identifiers are. I did this using four regular expressions inside the third TOKEN production



```
// Define what identifiers and numbers are
TOKEN : {
      < #DIGIT : ["0"-"9"] >
    | < NUMBER : ("-")* ["1"-"9"] (<DIGIT>)* | "0" >
    | < #LETTER : ["a"-"z", "A"-"Z"] >
    | < IDENT : <LETTER> (<LETTER>  | "_" | <DIGIT>)* >
}
```

From above, the hashtag infront of DIGIT means that digit is only used internally by JavaCC. It cannot be used as a token in the language. It is there to help define what a number is. The same applies to #LETTER

Syntactical Analysis

Overview

Syntax analysis is the second phase of the compiler. This phase deals with checking that the stream of tokens produced during the lexical analysis phase form valid 'sentences'. This is achieved by defining a set of production rules.

Implementation

This section deals with SECTION 4: THE GRAMMAR of the source file (CCALParser.jj)

In JavaCC production rules are defined similar to functions and are composed of regular expressions, other production rules and the tokens defined in SECTION 3.

I implemented the production rules as layed out in the CCAL language specification.

JavaCC generates top-down parsers that handle LL(1) grammars. Top down parsers start at the root of the parse tree labelled with the goal symbol of the grammar and repeat a set of operations until the fringe of the parse tree matches the input string.

In the grammar defined for the CCAL language, the `program` symbol is the start symbol.

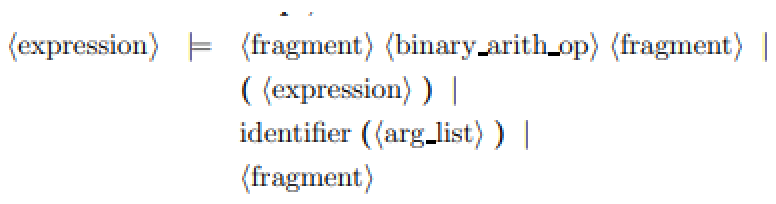
The grammar in this spec is ambiguous and is certainly not an LL(1) grammar, however it can be converted to one by manipulating the grammar according to certain rules.

The most noticeable ambiguous parts of the grammar were around the `condition` , `expression` , and `fragment` production rules as they contained left-factors and left recursion which top-down parsers can not handle.

Firstly, I wrote out these production rules on paper and used certain grammar manipulation rules to remove the left factors and left recursion.

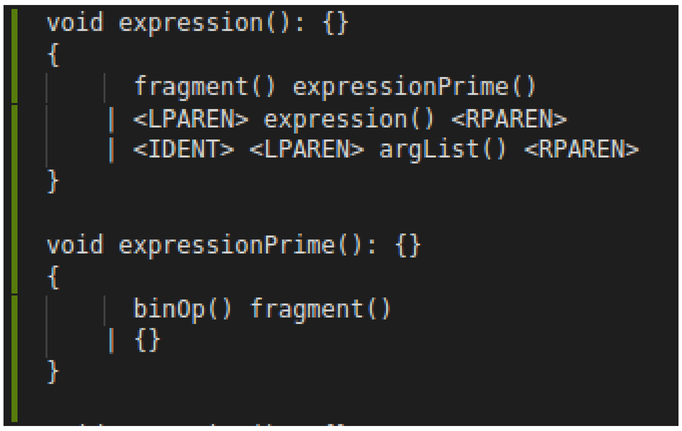
I started by removing any left factors in the grammar.

A before and after example of the `expression` production rule can be found below.



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

After removing the `fragment` left factor, I got:



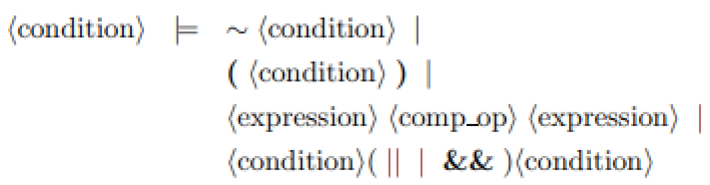
```
void expression(): {}
{
        fragment() expressionPrime()
      | <LPAREN> expression() <RPAREN>
      | <IDENT> <LPAREN> argList() <RPAREN>
}

void expressionPrime(): {}
{
        binOp() fragment()
      | {}
}
```

I carried out this process for all other productions that contained left factors including `statement` , `nempArgList` , and `nempParameterList` .

Next, I had to remove left recursion. I started by removing direct left recursion from production rules.

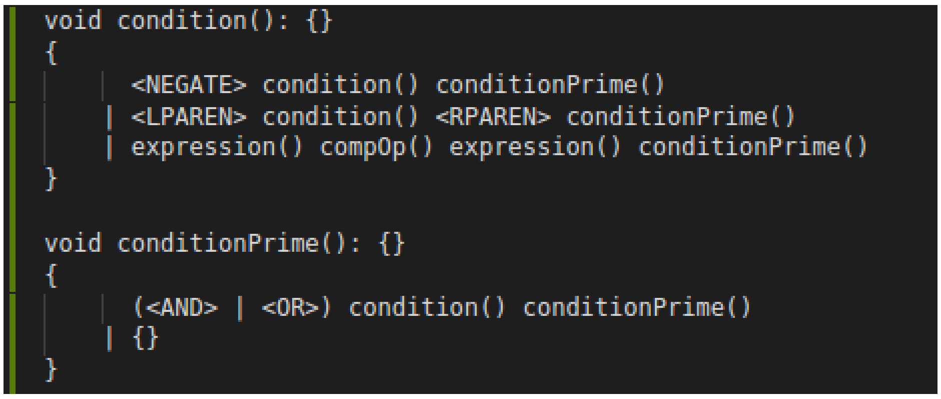
An example of this is the direct left recursion in the `condition` production rule.

Before:



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

After:



```
void condition(): {}
{
        <NEGATE> condition() conditionPrime()
      | <LPAREN> condition() <RPAREN> conditionPrime()
      | expression() compOp() expression() conditionPrime()
}

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

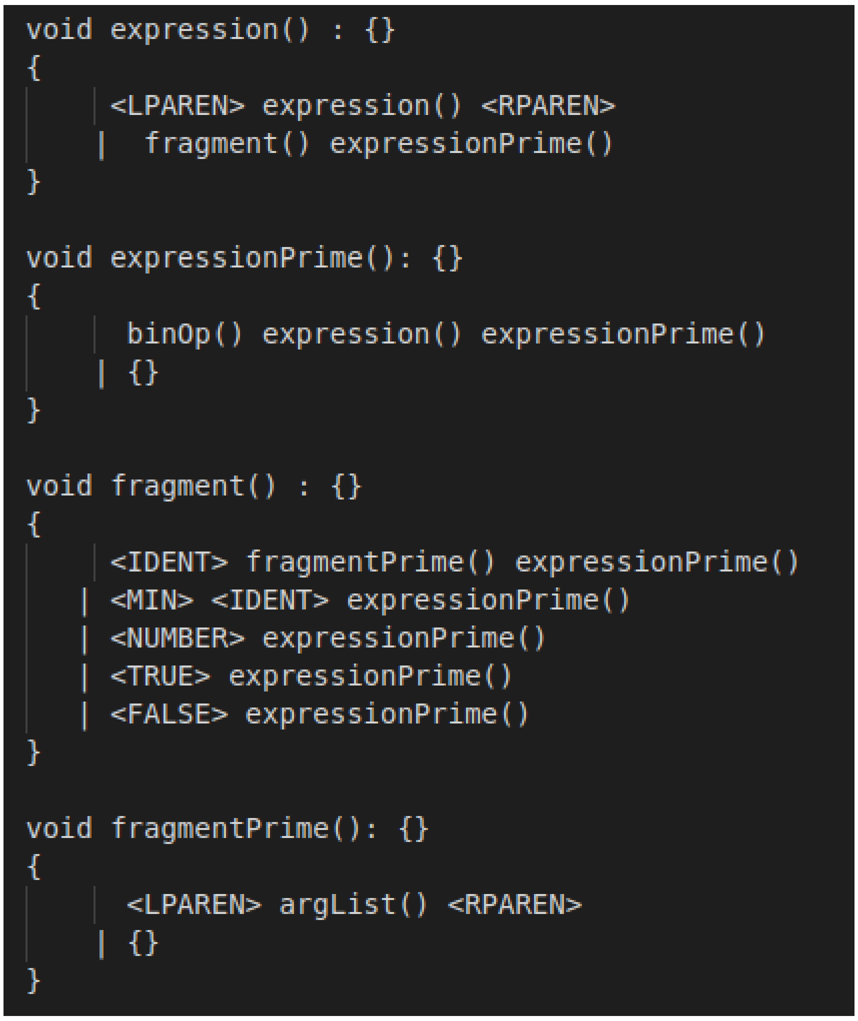
Now I faced another issue. This was the problem of indirect left recursion. I was not sure about how to do this with more complex production rules such as the ones in this grammar but after some searching I found out how.

Indirect left recursion existed between the `expression` and `fragment` production rules.

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

$$\begin{array}{lcl} \langle \text{fragment} \rangle & \models & \text{identifier} \mid \textbf{-} \; \text{identifier} \mid \text{number} \mid \textbf{true} \mid \textbf{false} \mid \\ & & \langle \text{expression} \rangle \end{array}$$

To remove the indirect left recursion I added a new non-terminal called `expressionPrime` . I was left with the following after removing the recursion:



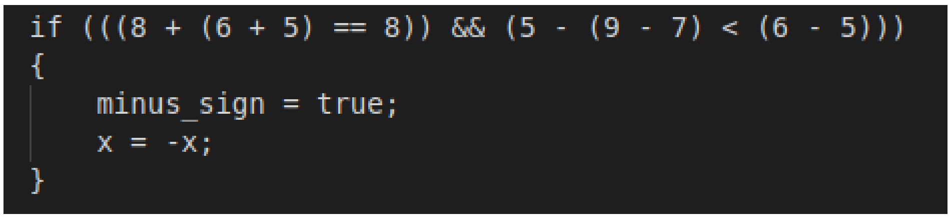
```
void expression() : {}
{
    <LPAREN> expression() <RPAREN>
    |  fragment() expressionPrime()
}

void expressionPrime(): {}
{
     binOp() expression() expressionPrime()
    | {}
}

void fragment() : {}
{
    <IDENT> fragmentPrime() expressionPrime()
   | <MIN> <IDENT> expressionPrime()
   | <NUMBER> expressionPrime()
   | <TRUE> expressionPrime()
   | <FALSE> expressionPrime()
}

void fragmentPrime(): {}
{
     <LPAREN> argList() <RPAREN>
    | {}
}
```

I was aware this section had to avoid using lookaheads. When all left factors and left recursion was eliminated, there was no need for lookaheads and the grammar was an LL(1) grammar.

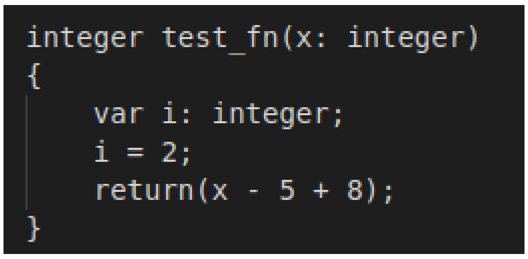
Testing

The approach I took to testing was looking at each production rule and writing out test CCAL programs to test each rule. There were production rules I focused on more such as `condition` , `expression` and `fragment` as these are the rules I had to manipulate the most to remove any left factors or left recursion.

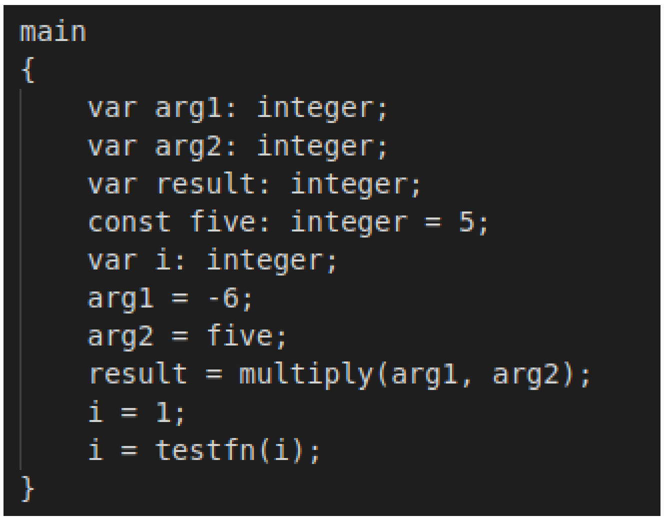
Below are some samples from those test programs to test whether I had implemented the grammar correctly.



```
if (((8 + (6 + 5) == 8)) && (5 - (9 - 7) < (6 - 5)))
{
    minus_sign = true;
    x = -x;
}
```



```
integer test_fn(x: integer)
{
    var i: integer;
    i = 2;
    return(x - 5 + 8);
}
```



```
main
{
    var arg1: integer;
    var arg2: integer;
    var result: integer;
    const five: integer = 5;
    var i: integer;
    arg1 = -6;
    arg2 = five;
    result = multiply(arg1, arg2);
    i = 1;
    i = testfn(i);
}
```

All my tests were passing as expected. I also wrote some test programs that I expected to fail. Each of the programs that I expected to fail the parsing did.

Conclusion

When I initially took a look at this assignment, it looked like it was going to be extremely difficult. However, as I broke the assignment down into its main components it became very easy to manage and wasn't so daunting.

I learned a lot from this assignment and I believe it's going to be a huge help during the final exam. I've also gained a big appreciation for people who design and develop compilers. I also learned a lot about regular expressions which was something I hadn't had much experience with prior.

Overall, this was a great assignment and has gave me a much better understanding of the module as a whole.

References

- http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node30.html
- https://www.gatevidyalay.com/left-recursion-left-recursion-elimination/
- http://www.d.umn.edu/~hudson/5641/l11m.pdf
- https://www.gatevidyalay.com/left-factoring-examples-compiler-design/
- https://javacc.org/