**CA4003 Assignment 1**

**Name(s):** Samuel Wood

**Programme:** Computer Applications, Year 4

**Module Code:** CA4003

**Assignment Title:** A Lexical and Syntax Analyzer for the CCAL Language

**Submission Date:** 2/ 11/ 2019

**Module Coordinator:** David Sinclair

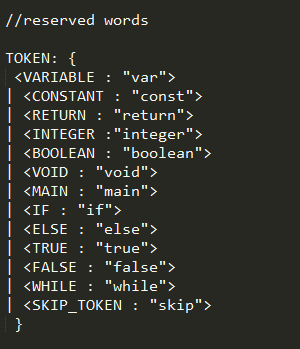
**Student Number:** 16476006

**Introduction**

The aim of this assignment is to implement a functional Lexical and Syntax Analyzer for the CCAL language specified in the assignment document. At the very beginning of this assignment I reviewed the documentation on the Javacc file. Given that I found this to be quite a challenging module I also watched a number of youtube videos on compiler design in general and using Javacc for a Lexical analyzer.

**Tokens**

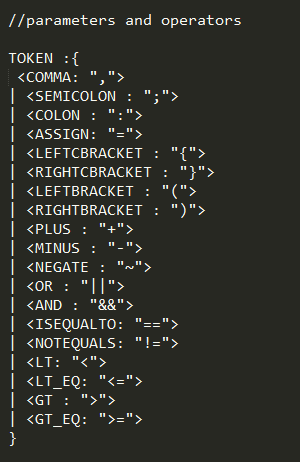
To implement the lexical and syntax analyzer I had to implement the reserved words in the language. As per the assignment definition these words were ‘var’, ‘ const’, ‘return’, ‘integer’, ‘boolean’, ‘void’, ‘main’, ‘if’, ‘else’, ‘true’, ‘false’, ‘while’ and ‘skip’. The following image is a snippet from my Javacc code that has all of the Reserved Tokens.



The rest of the required tokens for the CCAL language were as follows;

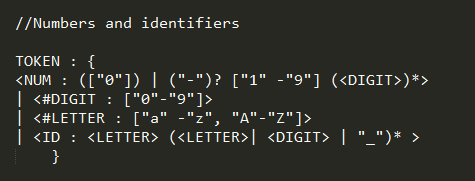
‘,’ , ‘;’ , ‘:’ , ‘=’, ‘{‘ , ‘}’ , ‘(‘ , ‘)’, ‘+’, ‘-’, ‘~’, ‘||’, ‘&&’, ‘==’, ‘!=’, ‘<’, ‘<=’, ‘>’, ‘>=’.

The next snippet is how they were implemented in the code.

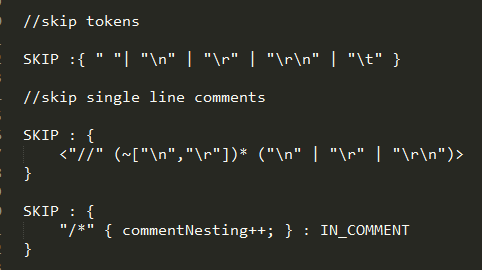


Now that the languages reserved words, operators, parameters and brackets etc had been defined, the next step was numbers and identifiers. The Digit and Letter definitions are to help produce other tokens. For example to define a negative number less than -9, there would need to be another digit after the first one, all of which is after the minus. The ‘\*’ symbolises that there can be more than one occurence of that token. The following is

the number and identifiers definitions from my code.

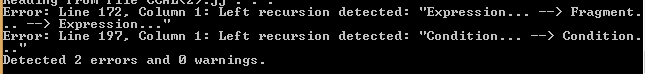


I had to use the skip function several times so the parser would operate correctly. For example blank spaces and new line characters. The CCAL language is also supposed to skip comments delimited by ‘/\*’ and ‘//’. The following is a snippet of where I said what needs to be skipped and also defined the comment delimiters. In the last fragment of this snippet you can see that when a ‘/\*’ is parsed,the commentNesting attribute is put to 1 and set back to 0 when a ‘\*/ is parsed. This is to change the state of the <IN\_COMMENT> token, which is used to ignore characters ie, treat as comments.



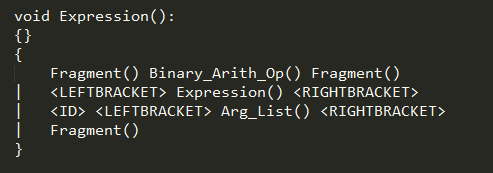
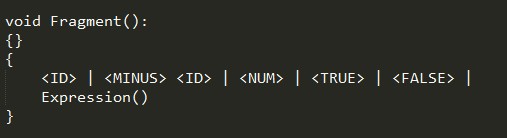
**Left Recursion**

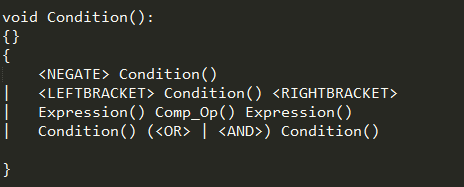
When I began working on the grammar I wrote out the functions exactly as they were specified in the assignment description. When I ran this grammar I received warnings that some of the elements in my grammar were left recursive.



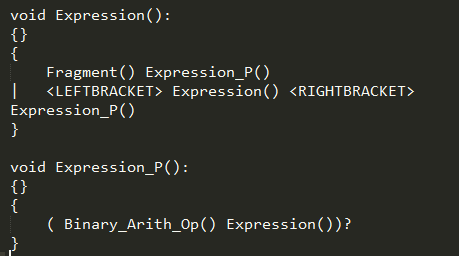
The problem methods with left recursion were Expression(), Fragment() and Condition(). The problem with the first two was that Fragment was called at the left most side in Expression and Expression was called in the left most side for the Fragment method. I was confused at first given that Expression is not directly called in itself and I received a left recursion error.

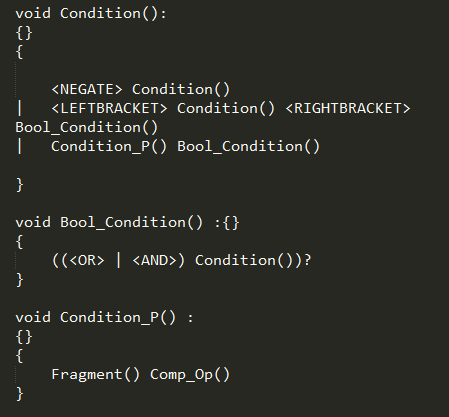
The problem with Condition was that it had called it self at the left most side. This is what the methods had looked like when I Received the error.





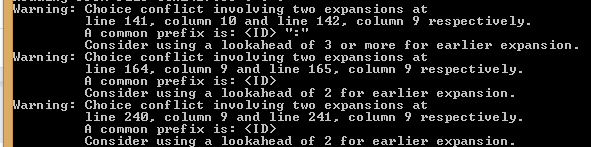
After reviewing the course notes and multiple videos on youtube regarding the topic I was able to eliminate the left recursion errors by adding additional functions and altering the existing ones. These additional functions were named ‘Expression\_2()’, ‘Condition\_P()’ and ‘Bool\_Condition()’. The following snippets are of the newly constructed methods and the changed already existing ones.



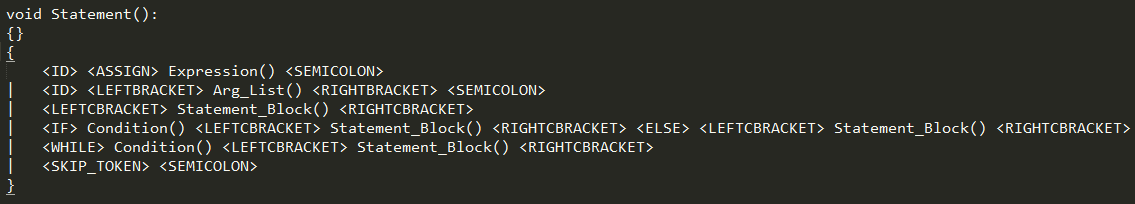


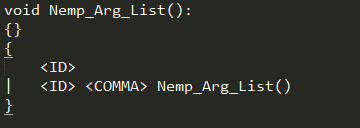
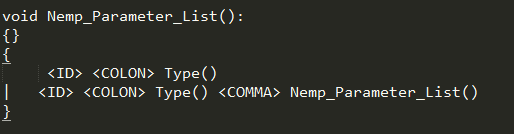
**Choice conflicts**

After the left recursion errors were no longer appearing I began to receive choice conflict warnings which advised me to use lookaheads in these functions. The functions that had these conflicts were ‘Nemp\_Parameter\_List()’, ‘Statement()’ and ‘Nemp\_Arg\_List()’. Below is a snippet of the choice conflict warnings in my command prompt.

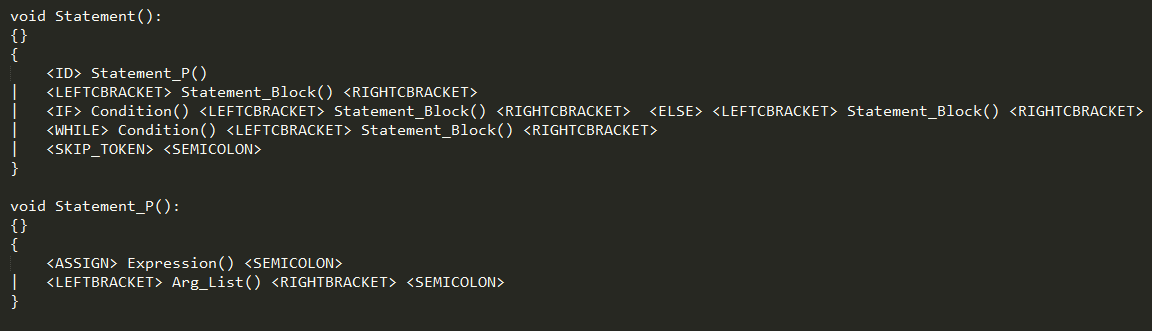


Before I began to look into choice conflicts and left factoring this is what the functions giving me the errors looked like:

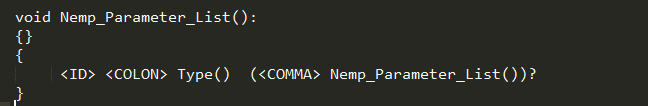


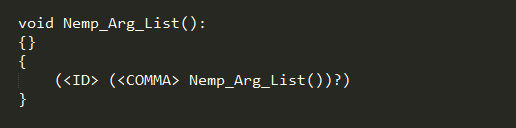


For ‘Statement()’, the issue involved the <ID> token. After further review of the course notes and additional study of Youtube videos on choice conflicts and left factoring, I implemented the new method Statement\_P() and altered the Nemp\_Parmeter\_List() and Nemp\_Arg\_List(). The new Statement method and the new Statement\_P() method were as follows.

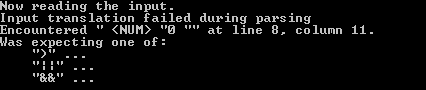


In these new methods the decision making has been postponed until the next method. In other words the first ‘Statement()’ method was non-deterministic and in the above snippet it is transformed to deterministic. In essence the function has just been re written. A similar approach to choice conflicts was taken to solving the other 2 methods, only for them a new method wasn’t created. They can be seen in the snippets below.

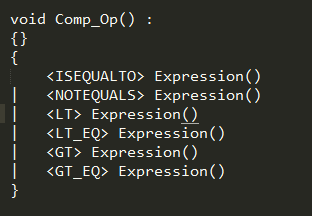




After having gotten this far,all of the test cases were passing except for the final one. Whenever I would run that text file I would get this error;



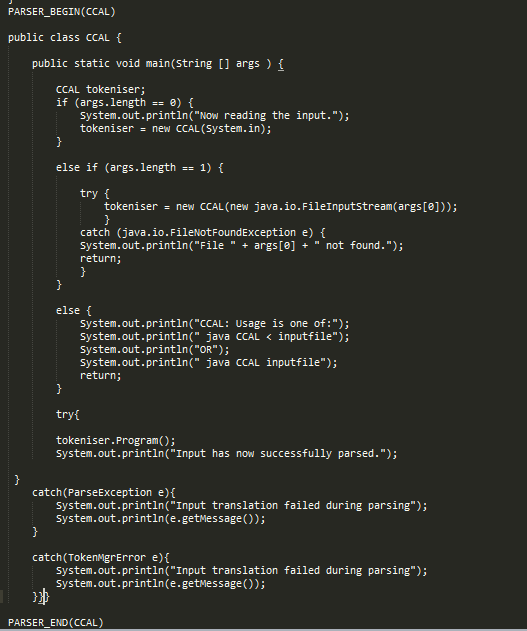
It came to my attention that this error came after a ‘>’ in the text file. After some experimenting I concluded that this error could be solved by adding an ‘Expression()’ method after the comparison operators in the ‘Comp\_Op()’ method. This is what the new ‘Comp\_Op()’ method looked like.



**Source Code**

During my time working on this assignment I had also recalled seeing the lexical analyzer source code. After going through this code from the notes myself I implemented it in the code, putting in some of my own messages. Below you will see an image of my source code.

Above this is an options section where the ‘IGNORE\_CASE’ variable is set to true as the case for the CCAL language is insensitive.



**Conclusion**

After the change to that method had been mad all test cases for the CCAL language that were given in the assignment spec had passed. I think this was one of the more enlightening assignments I have been given during my time in DCU. Having found the material covered in lectures quite abstract and difficult to grasp, this piece of work helped me get my head around a lot of it. Although this was a challenging piece of work, I am looking forward to working on the next stage of the CCAL compiler. I would also say that these 2 assignments will act as good revision for this modules exam come January.

**Sources of information used**

[**https://javacc.org/doc**](https://javacc.org/doc)

[**https://www.youtube.com/channel/UCJjC1hn78yZqTf0vdTC6wAQ**](https://www.youtube.com/channel/UCJjC1hn78yZqTf0vdTC6wAQ)

**https://www.youtube.com/channel/UCY4HMZumNbf4wcgPTdDpZlA**