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

**Declaration on Plagiarism**

**This form must be filled in and completed by the student(s) submitting an assignment**

|  |
| --- |
| **Name:** Emily McGivern |
| **Programme:** CASE 4 |
| **Module Code:** CA4003 |
| **Assignment Title:** Assignment 1 - A Lexical and Syntax Analyser for the CCAL Language |
| **Submission Date:** 3/11/2019 |
| **Module Coordinator:** David Sinclair |

**I declare that this material, which I now submit for assessment, is entirely my 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 work. I 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 have read and understood the Assignment Regulations. I 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 or any other person for assessment on this or any other course of study.**

**I 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:** Emily McGivern **Date:** 3/11/2019

**Introduction**

The aim of this assignment was to implement a Lexical and Syntax analyser using JavaCC for a simple language called CCAL. I started by using the code from the notes to begin my parser. This code served as a skeleton that I could build on further.

**Main method**

The main method which I implemented is largely from the code in the notes. I named the parser *CCALParser* to reflect the assignment goal. The parser is initialised here with an argument. It reads input from standard input, or it allows for a file to be passed as an argument. If the file that is passed in is not found, it returns a file not found error. After the parser is initialised, it will call the program() method and begin to parse input. If the file can be parsed correctly, it will return a message telling the user. If it cannot be parsed correctly, it will tell the user and return the specific error message.

**Options**

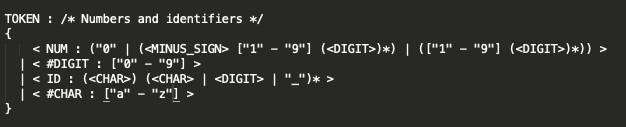
Within the options, I included the IGNORE\_CASE option and set it to *true*. This is because the language definition states it is not case sensitive. By default, this option is set to *false* which is why it needed to be included.

**Token Definitions**

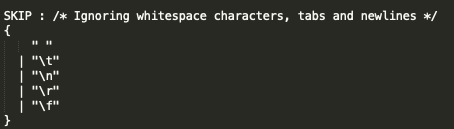
I defined the tokens according to the language definition provided. The tokens I included were reserved words and other tokens. Reserved words were words like **var, const, return and boolean**. Other tokens included **comma(,), semi-colon(;) and brackets**

**( { , ( , } , } )**. Further token definitions made were **numbers, digits, identifiers and characters.**

* **Numbers:** In this language, numbers are defined as being “represented by a string of one or more digits (‘0’- ‘9’) that do not start with the digit ‘0’, but may start with a minus sign (‘-’), e.g. 123, -456”. To deal with numbers in the language, I created a regular expression that would handle all possible combination of numbers which are accepted by the language.
* **Digits:** As digits were defined as ‘0’ - ‘9’ in the language, I included this definition in my tokens.
* **Identifiers:** In this language, identifiers are defined as being “represented by a string of letters, digits or underscore character (‘\_’) beginning with a letter. Identifiers cannot be reserved words”. To deal with identifiers in the language, I used another regular expression to ensure that all possible valid combinations would be accepted.
* **Characters:** I included a simple definition of a character to accept any characters from ‘a’ - ‘z’.



* **Whitespace, tabs and newlines**: These could be ignored by the parser. This was done by telling the parser to skip them when they occurred by placing them in a region called “SKIP”.



* **Comments:** Comments also had to be dealt with using SKIP. The language allows nested comments and comments that cannot be nested. Nested comments are delimited by /\* and \*/. Comments that cannot be nested begin with // and are delimited by an end of line. To handle the nested comments, a counter had to be used to keep track of the beginning and ending of each comment. When a /\* occurred, the counter incremented by 1, when a \*/ occurred it decremented by 1. This is to ensure that the nested comments are balanced. The regular expression for single-line comments checks for a // at the beginning and a new-line character for the ending with anything occurring in between these.

**Grammar**

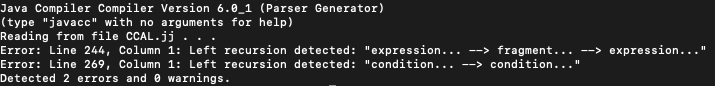
When defining the grammar at first, I wrote it as it appeared in the language definition. This

made it easier to ensure that I had all required rules covered in my grammar, and could change them as required later on. When I compiled the file after finishing the grammar, a number of errors related to left recursion occurred.

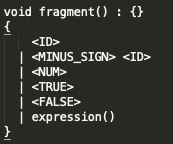
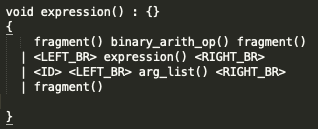
**Left Recursion**

As JavaCC does not allow for left recursion, the left recursion in my code had to be removed before I could continue.

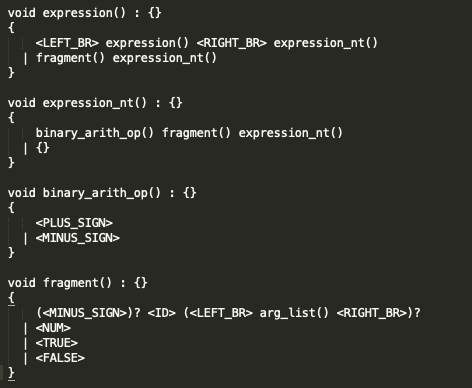
The left recursion errors which arose were as follows:



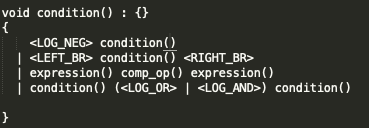
The first error was caused by the following blocks of code:



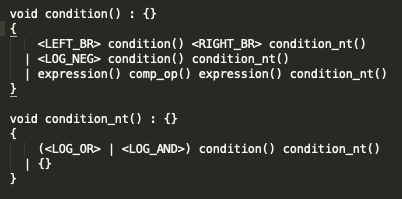
This is an example of indirect left-recursive grammar as expression… -> fragment… -> expression… ->. Expression() calls fragment() and at the end of fragment(), expression() is called again. To eliminate the left-recursion, I had to work on transforming the grammar. I removed expression() from the left-hand side of the production rule in fragment(). I also created the method expression\_nt() which evaluates binary\_arith\_op() fragment() expression\_nt() or it evaluates to nothing.



The second left recursion error was caused by this block of code:



In this case, the left recursion was caused by condition() (<LOG\_OR> |< LOG\_AND>) condition(). As before, I had to transform the grammar to eliminate the left-recursion. I created a new method called condition\_nt() which handles the AND and OR with the methods to ensure that recursion doesn’t occur. The following shows the changes I made:

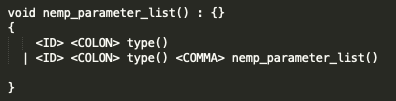


After dealing with the left recursion errors, I then compiled the parser again. I was then presented with a number of choice conflicts.

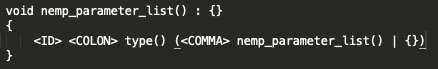
**Choice Conflicts**

The majority of the choice conflicts which arose were related to <ID>. These were easily solved by using left factoring. The following are examples of some of the changes made to pieces of code to remove the choice conflicts.

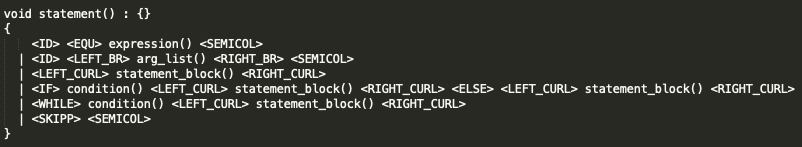
**Before**



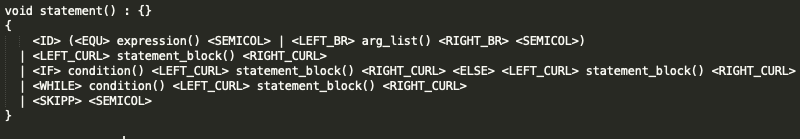
**After**



**Before**



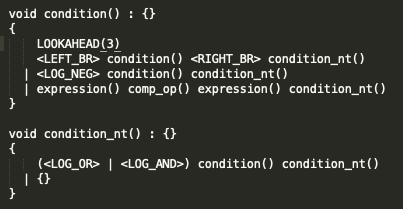
**After**



Another choice conflict which arose was in condition(). It was shown as follows:



This choice conflict was related to a left bracket. I could not solve it using the methods I had previously used to solve the others. I chose to implement a lookahead of 3 as suggested. This resolved the choice conflict.



After I had resolved all errors and messages in the terminal, I compiled it once again. It compiled with no errors.

**Building and Testing**

Once I had completed the parser, I needed to test it to ensure that it would correctly parse the CCAL language. To do this, I create four different test files. I got the code for these test files from the language definition provided.

Steps

1. Compile the parser with JavaCC *javacc CCALParser.jj*
2. Compile the corresponding Java file with Javac *javac CCALParser.java*
3. Run with test files *java CCALParser test1.txt*