# Declaration on Plagiarism

## Assignment Submission Form

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

| |
|---|
| Name(s): Toba Toki |
| Programme: CASE4 |
| Module Code: CA4003 |
| Assignment Title: Assignment 1: A Lexical and Syntax Analyser |
| Submission Date: 12/11/18 |
| Module Coordinator: David Sinclair |

I/We 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(s): _____Toba Toki_____ Date: ____12/11/18_____

The aim of this report is to describe the process I undertook to create a lexical and syntax analyser for CAL language.

Firstly, we begin with options. As per the instructions given, the language was to be case insensitive. This means that I had to treat or interpret upper- and lowercase letters as being the same. For me to do this, for my options, I had to make “IGNORE_CASE = true”.

```
options {

IGNORE_CASE = true;

 }
```

To begin a javacc parser, it must first of all begin with a declaration of its name. For mine it was “PARSER_BEGIN(cal)”.

```
PARSER_BEGIN(cal)
```

After this I had to initialise the parser. This is so that if the input file of the language code was passed as a command line argument, it would run and if not, it would throw an exception. I used this using “try and catch”. If the program successfully parses a test that you have given it, it would display “Parse Successful”. It would also be able to throw up token errors and parse exceptions. If peradventure the file is not found, it would return an error telling you it is not found.

```
  try {
    cal parser = new cal(new java.io.FileInputStream(args[0]));
    parser.program();
    System.out.println("Parse Successful!");
   }
   catch (ParseException e) {
       System.err.println("Parse Exception");
       throw(e);
   }
    catch (TokenMgrError e) {
   System.err.println("Token Error");
    throw(e);
   }

   catch (FileNotFoundException e) {
       System.err.println("File " + args[0] + " not found.");
  }
 }
}
```

At the end of this, I closed the parser off with its declaration.

```
PARSER_END(cal)
```

Next thing I had to do was the definition of all the tokens given to us.  I created a regular expression

```
SKIP:
{
    < " " | "\t" | "\n" | "\r" | "\r\n" >
    | < "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n") >
    | < "/*" (~[])* "*""/" >
}
```

to deal with any type of whitespace or comments. What it does is that is ignores spaces, tabs as well as newlines. In order for the “SKIP” regular expression to find spaces in single line comments, I changed the regular expression around so that it would be able to deal with ASCII characters that ranged from space to tilde.

Then I created and defined all the operators that would be used in this language.

```
TOKEN : /* OPERATORS */
{

  < COMMA : "," >
| < SEMI_COLON : ";" >
| < COLON : ":" >
| < ASSIGN : ":=" >
| < LBR : "(" >
| < RBR : ")" >
| < ADD : "+" >
| < MINUS : "-" >
| < NEGATE : "~" >
| < OR : "|" >
| < AND : "&" >
| < EQUALS_TO : "=" >
| < NOT_EQUALS_TO : "!=" >
| < LESS_THAN : "< " >
| < LESS_THAN_EQUALS : "<=" >
| < GREATER_THAN : ">" >
| < GREATER_THAN_EQUALS : ">=" >


}
```

I then created and defined the reserved words as instructed to do so.

```
//RESERVED WORDS
TOKEN :
{

  < VARIABLE : "variable" >
| < CONSTANT : "constant" >
| < RETURN : "return" >
| < INTEGER : "integer" >
| < BOOLEAN : "boolean" >
| < VOID : "void" >
| < MAIN : "main" >
| < IF : "if" >
| < ELSE : "else" >
| < TRUE : "true" >
| < FALSE : "false" >
| < WHILE: "while" >
| < BEGIN : "begin" >
| < END : "end" >
| < IS : "is" >
| < SKP : "skip" >

}
```

After doing all this, I decided to compile my parser. A ParseException errors was being thrown due to one of my token definitions.

```
org.javacc.parser.ParseException: Encountered " "SKIP" "SKIP "" at line 109,
column 5.
```

This was due to the definition of the reserved word token I had named “ `< SKIP >` ”. What was happening was that it was conflicting with the regular expression I had made earlier for skipping spaces, tabs and new lines. For me to fix this, all I had to do was to rename the token. I renamed it to `< SKP >`.

When I was doing the tokens for characters, digits, numbers and identifiers, I found out that the way I went about doing my “NUMBER” was wrong. After reading the instructions in great detail, I learned that for the CAL language, “NUMBER” could not begin with a 0 but that they could be negative. Before I read it properly, it didn’t come to my attention that “NUMBER” could begin with 0. This was only possible if the number began with a 0 and no digits succeeding it. Thus, the number could only be 0. This was why for number, I added an or to say that a number could simply be 0 if the user wanted it to be.

```
TOKEN :
{

  < #CHAR : [ "a"-"z" ] | [ "A"-"Z"] >
| < #DIGIT : ["0"-"9"] >
| < NUMBER : ("-")?["1"-"9"](< DIGIT >)*| "0" >
| < IDENTIFIER : (< CHAR >) ((< DIGIT >) | (< CHAR >) | "_")* >

}
```

Now it was time for me to do the grammar rules. After understanding what each of the functions were and what they were meant to do, I was able to modify some of the rules due to compiling errors from running the parser program.

Using Kleene closure, I was able to change some of the rules such as “ decl_list() ”. “ decl_list() ” was not a very important function as all it did was add a “ < SEMI_COLON > ” to any number of calls that “ decl() ” made. This was why I transferred the “ < SEMI_COLON > ” to the end of the “ decl() ” rules. I then replaced anywhere I saw “ decl_list() ” to “( decl() )* ”.

Not only did I do this just for “ decl_list() ” and “ decl() ”. I also did thus for “ nemp_parameter_list() ” and “ nemp_arg_list() ”.

What “ nemp_parameter_list() ” does is that it lists one or more identifiers as parameters when it came to declaring a function. “ nemp_parameter_list() ” calls itself so that I’m able to call

```
<IDENTIFIER> <COLON> type()
```

Again. For this reason, I changed the rule to

```
void parameter(): { }
{
    <IDENTIFIER> <COLON> type() ( <COMMA> < IDENTIFIER > < COLON > type() )*
}
```

As well as changing the name to “ parameter() ”. This then meant that whenever I came across “ parameter_list() ”, I was able to replace it with “ ( parameter() )* ”.

I went on to do the same to " nemp_arg_list() ". As well as doing the same to " function_list() " and " function() ".

A left-recursion problem was detected when I tried to compile the parser program.

```
Error: Line 206, Column 1: Left recursion detected: "statement_block... -->
statement_block..."
```

Using Kleene closure again we were able to remove this left recursion problem. The Kleene closure of " statement() " replaced the function " statement_block() " because it was no longer needed.

There was also a left recursion problem in the " condition() " rule.

```
Error: Line 286, Column 1: Left recursion detected: "condition... -->
condition..."
```

Lucky for me, I figured out that the " comp_op() " function was only being called in " condition() ". Hence, I was able to exploit this because other functions and other rules would not be affected if I changed around the " comp_op() ". I added " < OR > | < AND > " token into " comp_op() ". I added the " condition() " call into the " comp_op " rule.

```
void comp_op(): { }
{
  ( < EQUALS_TO > condition() | < NOT_EQUALS_TO > condition()
| < LESS_THAN > condition() | < LESS_THAN_EQUALS > condition()
| < GREATER_THAN > condition() | < GREATER_THAN_EQUALS > condition()
| < OR > condition() | < AND > condition() | {} )
}
```

In the " expression() " rule, left recursion also appeared for it.

```
Error: Line 252, Column 1: Left recursion detected: "expression... -->
fragment... --> expression..."
```

It was very easy to solve this as all I had to do was add my " < LBR > " and " < RBR > " tokens to the " expression() " call in the " fragment() " function.

```
void fragment(): { }
{

   ( < IDENTIFIER > | < NUMBER > | < TRUE > | < FALSE > | < LBR > expression() < RBR >
}
```

I had exactly 5 choice conflicts throughout creating the parser. I solved two of them using Kleene closure as mentioned above. The two of these choice conflicts that I solved were " nemp_arg_list() " as well as " nemp_parameter_list() ". The other three choice conflicts I had that I wasn't able to solve were " expression() ", " condition() " and " statement() ". For these three choice conflicts, I used " LOOKAHEAD(2) ".

When I tried to parse the examples given in the instructions, the first three tests parsed successfully. Nevertheless, the other examples did throw some parse exceptions.

For the fourth test file, when I copied and pasted the “A simple file demonstrating comments”, and I ran it, I got a strange error. I then noticed that the characters of the asterisk for comments I copied from the pdf was different to the one I had made for the “SKIP” rule. To fix this I just had to change the asterisk to the correct one.

For the fifth test file, my program encountered a ParseException where it encountered a semicolon. Since the parser was not expecting a semicolon on the line “ func() ”, I had to fix this by adding “ < SEMI_COLON > ” to the “ statement() ” rule. This would then allow the program to be able to identify and accept the semicolon at the end of the “ func() ” line.

For the sixth test file, I didn’t experience any errors meaning I can assume that if there was any issues that it should have had, it was fixed by the solutions I had used previously to getting to this test.

For the seventh test file, I had the same “TokenMgrError: Lexical error” thrown because like the test fourth file, me copying and pasting the examples changed how some of the tokens were represented. The minus character that was in front of the “x” in the line “ x := -x ” was different to the minus sign I used for my operator. I encountered a ParseException for skip but I fixed it by adding `< SKP > < SEMI_COLON >` to the “ statement() ” rule.

To run my program:

- Javacc cal.jj
- Javac *.java
- Java cal test”whatever test number you want”.cal