## Declaration on Plagiarism

## Assignment Submission Form

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

Name(s): Killian Byrne

Programme: *Compiler construction*
Module Code: *CA4003*
Assignment Title: *Assignment 1: A Lexical and Syntax Analyser for the CCAL Language*
Submission Date: 04/11/2019
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):Killian Byrne Date:03/11/2019

# Introduction:

The purpose of this report is to outline how I obtained my final grammar which is found in “assignment_one.jj”. There are four sections found in my grammar, options, user code, token definitions, and grammar. I will begin my describing each section of my jj file.

# Options:

For this section, we only needed one option, “ignore_case = true”. The need for this option was because the specifications for this language requires it to not be case dependent. This was found in the ccal.pdf file.

# User code:

Javacc parsers require a declaration of the parser name, in our case it is called by:

- PARSER_BEGIN(assignment_one)

Next we need to provide code to ensure that will initialise the parser if the input file of the language code is passed as a command line argument. If this condition is not met, exit with an instruction on how to run the parser.

```
public class assignment_one {
    public static void main(String[] args) {
        assignment_one tokeniser;

        // Initialise parser
        if(args.length == 1) {
            try {
                tokeniser = new assignment_one(new FileInputStream(args[0]));
            }
            catch(FileNotFoundException e) {
                System.err.println("File " + args[0] + " not found");
                return;
            }
        } else {
            System.out.println("assignment_one can be used by entering the following command:");
            System.out.println("     java assignment_one inputfile");
            return;
        }
```

Next, we need to display the contents of the file by each token and its corresponding lexeme. A hash map was a suitable option here as it simple to read from and it improves efficiency.

```
    HashMap allTokens = new HashMap();
    allTokens.put(VAR, "VAR");
    allTokens.put(CONST, "CONST");
    allTokens.put(RETURN, "RETURN");
    allTokens.put(INTEGER, "INTEGER");
    allTokens.put(BOOLEAN, "BOOLEAN");
    allTokens.put(VOID, "VOID");
    allTokens.put(MAIN, "MAIN");
    allTokens.put(IF, "IF");
```

Our next step is to parse the program. As seen below, a try and catch block is required for the possibility of an error. The parser is initialized here because the tokens are displayed above this block of code, and if it wasn't initialized the program would try to read the tokens downwards from this block of code.

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

# Token definitions:

This section was straight forward, as the required token definitions are given in the ccal.pdf file. Firstly, we must ignore comments which can be either "//" or "/*", and whitespaces.

Next we supply the list of tokens that are required for this language, that again were found in the ccal.pdf page: var,const,return,integer,boolean,void,main,if,else,true,false,while and skip. We then go on to also list the symbols and operators. Identifier tokens and number tokens are also required.

```
/* Ignore any whitespace */
SKIP : {
    "\t"
  | "\n"
  | "\r"
  | "\f"
  | " "
}

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

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

# Grammar and production rules:

I began this section by simply following the grammar supplied on the ccal.pdf page. Once every grammar had been defined, it was then time to remove errors and warnings. The errors and warnings I encountered are as follows:

**Left-recursion problem 1**:

"*Error: Left recursion detected: expression…→ fragment…→ expression"*

Code:

```
void expression () : {}
{
// fragment call
        (fragment()) ( (binary_arith_op()) (fragment()) ) |
        <LEFT_BRACKET> (expression()) <RIGHT_BRACKET> |
        <IDENTIFIER> <LEFT_BRACKET> (arg_list()) <RIGHT_BRACKET> |
// fragment call
```

```
        (fragment())
}

void fragment () : {}
{
        <IDENTIFIER> |
        <MINUS> <IDENTIFIER> |
        <NUMBER> |
        <TRUE> |
        <FALSE> |
// expression call
        (expression())
}
```

Problem: The problem here was indirect left recursion. It can occur when we have non terminals $A_0$ , $A_1$ , …. , $A_n$ and $\alpha_1$ , $\alpha_1$ , …, $\alpha_1$ are a mix of non terminals and production rules are as follows:

$$\begin{aligned} A_0 &\rightarrow A_1\alpha_1 \mid \ldots \\ A_1 &\rightarrow A_2\alpha_2 \mid \ldots \\ &\ldots \\ A_n &\rightarrow A_0\alpha_{n+1} \mid \ldots \end{aligned}$$

This can lead to the leftmost derivation of the form:

$$A_0 \rightarrow A_1\alpha_1 \rightarrow A_2\alpha_2\alpha_1 \rightarrow \ldots \rightarrow A_0\alpha_{n+1}\ldots\alpha_2\alpha_1$$

This leads to problems for a top down parser. To fix this problem, we apply the strategy as follows:

- Let $\alpha$, $\boldsymbol{\beta}_1$ , $\boldsymbol{\beta}_2$ , … , $\boldsymbol{\beta}_n$ be a mix of terminal and non terminals.
- Arrange all non terminals into some arbitrary order.
- For each terminal $A_i$ , such that $1 \leq i < j$ and we have a production rule of the form $A_i \rightarrow A_j\alpha$ where the $A_j$ productions are $A_j \rightarrow \boldsymbol{\beta}_1 \mid \ldots \mid \boldsymbol{\beta}_n$ , do:
    - Replace the production rule $A_i \rightarrow A_j\alpha$ with the rule $A_i \rightarrow \boldsymbol{\beta}_1\alpha \mid \ldots \mid \boldsymbol{\beta}_n\alpha$ .
    - Eliminate any immediate left recursion among the $A_i$ productions.

Solution:

```
void expression() : {} {

    fragment() (
        binary_arith_op() fragment()
        | {}
    )
}

void fragment() : {} {
    <IDENTIFIER> (<LEFT_BRACKET> arg_list() <RIGHT_BRACKET> fragment_dash()| fragment_dash())
    | <MINUS> (<IDENTIFIER>|<NUMBER>) fragment_dash()
    | <NUMBER> fragment_dash()
    | <TRUE> fragment_dash()
    | <FALSE> fragment_dash()
    | <LEFT_BRACKET> expression() <RIGHT_BRACKET> fragment_dash()

}

void fragment_dash() : {} {

    binary_arith_op() fragment() fragment_dash()
    | {}
}
```

**Left-recursion problem 2**:

"*Error: Left recursion detected: "condition... --> condition..."*"

Code:

```
void condition () : {}
{
        <NOT> (condition()) |
        <LEFT_BRACKET> (condition()) <RIGHT_BRACKET> |
        (expression()) (comp_op()) (expression()) |
        (condition()) (<OR> | <AND>) (condition())
}
```

Problem:

This problem was relatively straight forward. When such a production rule exists:

$A \rightarrow A\ \alpha/\boldsymbol{\beta}$

The following expansion will occur:

$A \rightarrow A\ \alpha/\boldsymbol{\beta}$

$A \rightarrow A\ \alpha/\boldsymbol{\beta}\ \alpha/\boldsymbol{\beta}$

$A \rightarrow A\ \alpha/\boldsymbol{\beta}\ \alpha/\boldsymbol{\beta}\ \alpha/\boldsymbol{\beta}$

$A \rightarrow A\ \alpha/\boldsymbol{\beta}\ \alpha/\boldsymbol{\beta}\ \alpha/\boldsymbol{\beta}$ ……

The strategy to solve this is as follows:

$A \rightarrow A\ \alpha/\boldsymbol{\beta}$

Becomes:

$A \rightarrow \boldsymbol{\beta}A'$

$A' \rightarrow \alpha A' \mid \{\}$

Solution:

- Applied left recursion rule as described above to original function.

```
void condition (): {}
{
        <NOT> (condition()) (condition_dash()) |
        <LEFT_BRACKET> (condition()) <RIGHT_BRACKET> (condition_dash()) |
        (expression()) (comp_op()) (expression()) (condition_dash())
}

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

**Error 3: Choice conflict**

*Warning: Choice conflict involving two expansions at line 250, column 28 and line 250, column 37 respectively. A common prefix is: "integer"*

Code:

```
void nemp_parameter_list () : {}
{
        <IDENTIFIER> (<COLON> (type() | type() <COMMA> nemp_parameter_list()))
}
```

Problem:

- The parser reaches a choice conflict when it gets to the first type() above. This occurs because the two production rules both have prefixes in common. This is known as the common prefix problem. The formula to solve this problem, is as follows:
  $A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$
  *Solve by:*
  $A \rightarrow \alpha A'$
  $A' \rightarrow \beta_1 / \beta_2 / \beta_3$

Solution:

```
void nemp_parameter_list (): {}
{
        <IDENTIFIER> <COLON> type() nemp_parameter_list_prime()
}

void nemp_parameter_list_prime (): {}
```

```
{
        (<COMMA> nemp_parameter_list())?
}
```

**Error 4: Choice conflict**

*Warning: Choice conflict involving two expansions at line 342, column 5 and line 343, column 5 respectively. A common prefix is: "(" "("*

Code:

```
void condition (): {}
{
        <NOT> (condition()) (condition_dash()) |
        <LEFT_BRACKET> (condition()) <RIGHT_BRACKET> (condition_dash()) |
        (expression()) (comp_op()) (expression()) (condition_dash())
}
```

Problem: Choice conflict occurs due to presence of left brackets. Unfortunately I could not diagnose this problem (after many hours), so I used a lookahead of 1 to fix this.

Solution:

```
void condition (): {}
{
        <NOT> (condition()) (condition_dash()) |
        LOOKAHEAD(1)<LEFT_BRACKET> (condition()) <RIGHT_BRACKET>
(condition_dash()) |
        (expression()) (comp_op()) (expression()) (condition_dash())
}
```

**Error 5: Choice conflict**

*Warning: Choice conflict involving two expansions at line 364, column 5 and line 365, column 5 respectively. A common prefix is: <IDENTIFIER>*

Code:

```
void nemp_arg_list () : {}
{
        <IDENTIFIER> |
        <IDENTIFIER> <COMMA> (nemp_arg_list())
}
```

Problem:

- Choice conflict occurred because Identifier was a common prefix to both. This was solved by refactoring the statement so that it could try to match id, comma, nemp_arg_list. But if it didn't match that, then it could still match id.

Solution:

```
void arg_list () : {}
{
        (nemp_arg_list())
}

void nemp_arg_list () : {}
{
        <IDENTIFIER> (<COMMA> (nemp_arg_list()))?
}
```

# Parsing problems:

Now that the compiler could run without errors or warnings, the next step was to test the grammar against the test files found in the ccal.pdf page. Again I will outline some of the errors I encountered (I emphasize only some because if I was to list them all you would grow old).

**Problem 1:**

Problems trying to get the first few tests to pass:

Error messages:

```
For the following file...
MAIN ('MAIN') LEFT_BRACE ('{') RIGHT_BRACE ('}')
-----------------------------------------
There was an error while parsing:
ParseException: Encountered " "main" "MAIN "" at line 1, column 1.
Was expecting one of:
        "var" ...
        "const" ...
        "integer" ...
        "boolean" ...
        "void" ...
```

&

*"For the following file...*
*MAIN ('MAIN') LEFT_BRACE ('{') RIGHT_BRACE ('}')*
*----------*

*There was an error while parsing:*
*ParseException: Encountered " "}" "} "" at line 3, column 1.*
*Was expecting one of:*
*"var" ...*
*"const" ...*
*"if" ...*
*"while" ...*
*"skip" ...*
*"{" ...*
*<IDENTIFIER> …"*

Code:

```
void program (): {}
{
	(decl_list()) (function_list()) (main())
}

void decl_list (): {}
{
	(decl() <SEMICOLON> decl_list())?
}

void decl () : {}
{
	var_decl() | const_decl()
	//( decl() <SEMICOLON> decl_list() | {})
}

void var_decl () : {}
{
	<VAR> <IDENTIFIER> <COLON> type()
}

void const_decl () : {}
{
	<CONST> <IDENTIFIER> <COLON> type() <EQUALS> expression()
}
```

Solution:

```
void program() : {}
{
   ( decl() )*
   ( function() )*
   main()
}

void decl() : {}
{
   ( var_decl() | const_decl() )
}

void var_decl() : {}
{
   <VAR> ident_list() <COLON> type() ( <COMMA> ident_list() <COLON> type() )* <SEMICOLON>
}

void const_decl() : {}
```

```
{
   ( <CONST> <IDENTIFIER> <COLON> type() <EQUALS> expression() ( <COMMA> <IDENTIFIER> <COLON> type()
<EQUALS> expression() )* <SEMICOLON>)
}
```

Parsing problem 2:

Code:

```
//fix for parsing problem
/*void arg_list () : {}
{
        <IDENTIFIER> (<COMMA> (arg_list()))?
}*/
```

Solution:

```
void arg_list() : {}
{
   <LEFT_BRACKET> (<IDENTIFIER> ( <COMMA> <IDENTIFIER>)*  | {})
<RIGHT_BRACKET>
}
```

**Parser problem 3:**

This problem I encountered was one of a series of problems related to the statement block. I As seen in the code section, I had altered the grammar to fix the tests from the ccal.pdf file, but this last error stumped me and did not seem solvable to me. This resulted in having to rethink the structure of statement and start again. The crux of the problem seemed to revolve around the presence of (statement() <SEMICOLON>)*. This seemed to result in the parser expecting semicolons where it should not. I initially took this approach because of left recursion around the statement_block(). I tried a different approach and solved the left recursion without removing the statement block(). Once this was solved, the statement() grammar behaved as expected and passed the tests.

Error:

```
There was an error while parsing:
ParseException: Encountered " "if" "if "" at line 34, column 3.
Was expecting:
        ";" ...
```

Code:

```
void statement () : {}
{   <IDENTIFIER> statement_prime()
        | <LEFT_BRACE> statement_block() <RIGHT_BRACE>
        | <IF> condition() <LEFT_BRACE> statement_block() <RIGHT_BRACE>
        | <ELSE> <LEFT_BRACE> statement_block() <RIGHT_BRACE>
        | <WHILE> condition() <LEFT_BRACE> statement_block() <RIGHT_BRACE>
```

```
        | <SK><SEMICOLON>
}
```

ERROR:

There was an error while parsing:
ParseException: Encountered " "if" "if "" at line 13, column 8.
Was expecting:
        "{" ...

SOLUTION:

```
void statement_block () : {}
{
        (statement() statement_block())?
}

void statement () : {}
{   <IDENTIFIER> statement_prime()
        | <LEFT_BRACE> statement_block() <RIGHT_BRACE>
        | <IF> condition() <LEFT_BRACE> statement_block() <RIGHT_BRACE> <ELSE>
statement()
        //| <ELSE> <LEFT_BRACE> statement_block() <RIGHT_BRACE>
        | <WHILE> condition() <LEFT_BRACE> statement_block() <RIGHT_BRACE>
        | <SK><SEMICOLON>
}

void statement_prime () : {}
{
   <ASSIGNMENT> expression()<SEMICOLON>
   | <LEFT_BRACKET> arg_list() <RIGHT_BRACKET> <SEMICOLON>
}
```

# How to compile and run my parser:

In the winzip folder, I have included a bash script called run_all_tests.sh. The file is self explanatory. It simply automates the compiling and parsing for the ccal language and tests my grammar against the tests given on the ccal.pdf page. The file also echoes out every command that is run in the bash script.

- Run “bash run_all_tests.sh”

Or else:

- javacc assignment_one.jj
- javac *.java
- java assignment_one test_1.ccl (test files go from 1 - 7)