Declaration on Plagiarism

Assignment Submission Form

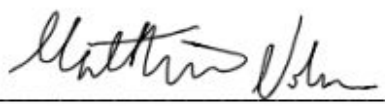
This form must be filled in and completed by the student(s) submitting an assignment

Name(s): **Matthew Nolan**
Programme: **Compiler Construction Assignment**
Module Code: **CA4003**
Assignment Title: **CCALL Language Parser**
Submission Date: **3/11/2019**
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 predicates, 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):  Date: 03/11/2019

CCAL Parser Description

In section one I defined two options for the parser; IGNORE_CASE and JAVA_UNICODE_ESCAPE. The IGNORE_CASE option was specified in the language definition as the CCAL language was not case sensitive and the JAVA_UNCODE_ESCAPE option was used to make sure the parser does unicode escapes the same was as a Java Compiler does. In section two I defined the tokeniser of the parser. For the user code I reused the code from the "JavaCC" Notes. I added these lines at the bottom to show whether or not the file was parsed correctly.

```
try {
    tokeniser.program();
    System.out.println("Parse successful");
} catch (ParseException e) {
    System.out.println(e.getMessage());
    System.out.println("Parse failure");
}
```

I used the default method for skipping newlines, spaces, tabs and mult-line comments that were in also in the notes. For single line comments I added the line : **<"//" (~["\n"])* "\n">.** This regular expression will find a string starting with "**//**" and ending with a newline character. Next I defined the keywords and tokens listed in the language definition. I had a minor issue here with the naming the token for the skip keyword as "SKIP" is already defined in JavaCC's token manager. Next I implemented the values recognised by the language.They were digits, numbers, letters and IDs. This is how I implemented them:

```
TOKEN : /* VALUES */
{
     <NUM : "0" | ((<MINUS>)? ["1" - "9"] (<DIGIT>)*) >
    |<#DIGIT: ["0"-"9"] >
    |<ID : <LETTER> (<LETTER> | <DIGIT> | <UNDERSCORE>)* >
    |<#LETTER: ["A"-"Z", "a"-"z"] >
}
```

Once I was finished with the tokens I implemented the grammar based off the definitions within the CCAL language definition. After I finished writing out the grammar I ran the code using the "javacc" command in the terminal.

Left Recursion

I got 2 warnings that there was left recursion present in the grammar. The first one was in the expression function; "expression → fragment → expression", the second one was in the condition function; "condition → condition". To fix left recursion in the expression function I put the grammar defined in the fragment function into the expression function and added a reference to expression within fragment. This made it easier to find the alpha and beta in the function in order to eliminate the recursion. I split the expression function into two separate functions. This eliminated the left recursion within the expression function. Similarly for the condition function I split it up into two functions but i had to add an expanded terminal function.

```
void expression() : {}
{
    choice_expression() expression_arith()
}

void choice_expression() : {}
{
     (<MINUS>)? <ID>  (LOOKAHEAD(1) <LBRACKET> arg_list() <RBRACKET>)?
    |<NUM>
    |<TRUE>
    |<FALSE>
    |<LBRACKET> expression() <RBRACKET>
}

void expression_arith() : {}
{
    binary_arith_op() expression()
    |{}
}

void fragment() : {}
{
    expression()
}
```

```
void condition() : {}
{
    choice_condition() condition_simple()
}

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

void condition_simple() : {}
{
     <TILDE> condition()
    |LOOKAHEAD(1) <LBRACKET>  condition() <RBRACKET>
    |expression() comp_op() expression()
}
```

Left Factoring

I noticed that i could factor a number of common prefixes throughout the grammar which I factored out. First and second line of the statement function both begin with "**<ID>**". I factored this by adding a separate function which is called after the "**<ID>**".

```
void statement() : {}
{
     <ID> choice_statement()
    |<LBRACE> statement_block() <RBRACE>
    |<IF> condition() <LBRACE> statement_block() <RBRACE> <ELSE> <LBRACE>
    statement_block() <RBRACE>
    |<WHILE> condition() <LBRACE> statement_block() <RBRACE>
    |<SKIPTOKEN> <SEMICOLON>
}

void choice_statement() : {}
{
     <ASSIGNMENT> expression() <SEMICOLON>
    |<LBRACKET> arg_list() <RBRACKET> <SEMICOLON>
}
```

Within the expression function I noticed two common prefixes.The lines "**<ID>**" and "**<MINUS> <ID>**" which I factored to "**(<MINUS>)? <ID>**". The other lines were"**<ID>**" and "**<ID> <LBRACKET> arg_list() <RBRACKET>**", which I factored to **"<ID> (<LBRACKET> arg_list() <RBRACKET>)?".** After factoring these lines I realised that I could factor them one more time into this:

```
void choice_expression() : {}
{
     (<MINUS>)? <ID>  (LOOKAHEAD(1) <LBRACKET> arg_list() <RBRACKET>)?
    |<NUM>
    |<TRUE>
    |<FALSE>
    |<LBRACKET> expression() <RBRACKET>
}
```

LookAheads

After the left recursion and left factoring had been resolved and completed, I had two choice conflicts due to the ambiguity of the language. After running the file with the javacc command it recommend to add a LOOKAHEAD(2) into the expression function and a LOOKAHEAD(3) into the condition function both due to a due to a common prefix of "(".

With proper placement of the lookaheads I only need a LOOKAHEAD(1) for each choice conflict.

```
void choice_expression() : {}
{
     (<MINUS>)? <ID>  (LOOKAHEAD(1) <LBRACKET> arg_list() <RBRACKET>)?
    |<NUM>
    |<TRUE>
    |<FALSE>
    |<LBRACKET> expression() <RBRACKET>
}
```

```
void condition_simple() : {}
{
     <TILDE> condition()
    |LOOKAHEAD(1) <LBRACKET>  condition() <RBRACKET>
    |expression() comp_op() expression()
}
```

This file is run by compiling the "**Assignment1.jj**" file with the javacc command. Then you must compile the java file with the Java compiler, compile the "CCALTokeniser.java" with javac command, then run the CCALTokeniser with the java command.