CA4003 - ASSIGNMENT 1:
A LEXICAL AND SYNTAX ANALYSER

STUDENT DETAILS

NAME: RUSSELL BRADY

STUDENT NUMBER: 15534623

PROGRAMME: CASE

MODULE CODE: CA4010

SUBMISSION DATE: 11/11/2018

MODULE COORDINATOR: DAVID SINCLAIR

PLAGIARISM DECLARATION

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/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 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: Russell Brady Date: 24/10/2018

TABLE OF CONTENTS

INTRODUCTION

In this report I will outline the major components, along with a description of my implementation, of my lexical and syntax analyser for the language CAL.

In generating the main JavaCC file there were four main sections with which to follow.

- Options
- User Code
- Token Definitions
- Grammar and Production Rules

Similarly, in creating this report, I have separated the description of my analyser into these four main sections.

OPTIONS

The grammar file starts with a list of options. Here allows us to specify code to set various options flags.

In our case we will only have one option. The only flag we wish to set is that we want our parser to be case insensitive. We do this like so:

```
options {
  IGNORE_CASE = true;
}
```

USER CODE

The user code is contained inside the following structure shown below:

```
PARSER_BEGIN(Assignment1)

public class Assignment1 {
    public static void main(String args[]) {
        //insert user code here
    }
}

PARSER_END(Assignment1)
```

As can be seen above, the code section must begin with PARSER_BEGIN and end with PARSER_END and both must have the parser name as a parameter, Assignment1 in this case. As well as this, a main method is included to read in an input file and to then run the program.

```
Assignment1 parser;

if ( args.length == 0 ) {

   System.out.println("Reading from standard input...");
   parser = new Assignment1(System.in);

} else if (args.length == 1) {
    try {

        parser = new Assignment1(new java.io.FileInputStream(args[0]));

    } catch (java.io.FileNotFoundException e) {

        System.err.println("File " + args[0] + " not found.");
        return;

    }
} else {

    System.out.println("Incorrect command entered. Assignment1 can be used with the following commands
    System.out.println("    java Assignment1 < file");
    System.out.println("    java Assignment1 file");
    return;

}
```

The code above deals with reading in an input file for parsing. It checks if an input file has been passed as a command line argument. If it has, then we will initialise the parser with this file. If no file has been passed, then the program will wait for the user to pass in the file from standard input. Once this is done the parser will then be initialised with this file. If there are more than one command line arguments, then the

program returns and prints out instructions for the user as can be seen above. If the file is not found, then the program exits displaying an appropriate error message to the user.

```
try {
    parser.Program();
    System.out.println("Program parsed successfully.");

} catch (ParseException e) {
    System.out.println(e.getMessage());
    System.out.println("Encountered errors while parsing.");
}
```

Once the parser is successfully initialised with the file, we then call parser.Program() to parse the file. This is wrapped in a Try / Catch block to catch a ParseException should It occur. If an error is caught, we know that the program is not valid CAL code. If the program parses successfully then the program prints a success message. If an exception is caught, then the error is returned to the user.

TOKEN DEFINITIONS

The language CAL is defined by a set of tokens in this section. Tokens are described by rules with the following syntax:

```
TOKEN : {
    <TOKEN_NAME : REGULAR_EXPRESSION>
}
```

The token name is the name of the token being described and the regular expression describes the token.

I separated this section up into different segments based on token type. I kept each type of token separate from the rest. For example, all the reserved word tokens were together, and all the operator tokens were together. I also created separate tokens for identifier and number which can be seen below:

```
//IDENTIFIER TOKEN
TOKEN : {
    <IDENTIFIER: <LETTER>(<LETTER>|<DIGIT>|<UNDERSCORE>)*>
    |  <#LETTER: (["a"-"z","A"-"Z"])>
    |  <#UNDERSCORE: (["_"])>
    |  <#DIGIT: ["0"-"9"]>
}

//NUMBER TOKEN
TOKEN : {
    < NUMBER : "0" | (<MINUS>)? ["1"-"9"] ((<DIGIT>)*) >
}
```

The number token was to be described as an integer which can be negative and must not have leading zero's. The number token uses a regular expression which is either zero or is a positive/negative number from 1 to 9 followed by any number of digits (with a digit being described as ["0" – "9"])

Another token of interest is the identifier token which describes the valid variable names in the language. This token uses a regular expression which says the identifier must start with a letter and can then be superseded by any number of letters, digits or underscores.

As well as defining the tokens of the language, this section also describes the values which may be skipped during parsing of an input file containing the CAL language. Some of the simpler values we wanted to skip included whitespace, tabs and newlines. However, we also wanted to be able to skip comments and these were a little bit trickier to describe.

For example, if we look at the second line in this SKIP we can see how single line comments are described:

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

With single line comments I described a regular expression with a double forward slash followed by any amount of characters in the ascii language and then ended by a newline. This ensured the comment only involved one line.

GRAMMAR AND PRODUCTION RULES

WRITING THE GRAMMER

I wrote the grammar for the language CAL based off the rules which were set out in the language definition. The initial grammar was very easily written by following the instructions of the language, however this was not going to suffice due to several problems which will be expanded upon below.
Once the initial grammar was written I then had to check that the grammar was working. I tried running the whole program however I was thrown a plethora of problems. I revaluated my approach and decided to comment out all non-terminal functions and recursively work my way up from the terminal functions. This proved to be a very quick and effective way of debugging the program and making the necessary changes.

```
void Type() : {}
{
    <INTEGER> | <BOOLEAN> | <VOID>
}
```

For example, I started with the non-terminal Type() above to ensure the program worked with just this function. Once I verified this I checked their parents and so on. Some of the errors I encountered as I checked each function are explained in the superseding sections.

LEFT RECURSION

I encountered left recursion twice while debugging my program. The first instance occurred where Expression() was calling Function() and then Function() was calling Expression(). This resulted in the following error:

Error: Line 234, Column 1: Left recursion detected: "Expression... --> Fragment...-->Expression"

The second instance involved Condition() where it was calling itself. This resulted in the following error:

Error: Line 262, Column 1: Left recursion detected: "Condition... --> Condition..."

Expanding on the second example, the code which caused this error can be seen below:

```
void Condition(): {}
{
    [<NOT>] Condition()
    |   <LEFTBRACKET> Condition() <RIGHTBRACKET>
    |   Expression() CompOp() Expression()
    |   Condition() (<OR> | <AND>) Condition()
}
```

We can see in the last line that Condition() is being called on the left hand side and this is causing the error. In order to fix this error, Condition() needed to be removed from the left hand side of the rule. To do this, I created Condition1(). I placed the rest of the rule without Condition() in Condition1() surrounded by [] to signify it can be empty and then placed Condition1() at the end of all the statements in Condition() to remove the left recursion. The result can be seen below:

```
void Condition(): {}
{
    [<NOT>] (<LEFTBRACKET> Condition() <RIGHTBRACKET> Condition1()
    | Fragment() [CompOp()] Condition1())
}

void Condition1() : {}
{
    [(<OR> | <AND>) Condition()]
}
```

KLEENE CLOSURE

After my initial grammar was written some of the production rules which were written were changed and transformed and in some cases I was able to get rid of some rules. This is known as Kleene Closure. It really helped to clean up my code and made it more concise, getting rid of redundant production rules.

An example of this is FunctionList() and Function().

```
void Program() : {}
{
    (DeclList())* FunctionList() (Main())*
}

void FunctionList(): {}
{
    (Function())*
}
```

```
void Program() : {}
{
    (Decl())* (Function())* Main()
}
```

Because FunctionList() is just zero or more instances of Function() I was able to replace FunctionList() with (Function())*. This makes the code a lot cleaner and less verbose.

Similarly DeclList() is equal to zero or more instances of (Decl() <SEMICOLON>). Therefore I was able to replace DeclList() with (Decl())* and move the <SEMICOLON> down into Decl(). This allows us to remove DeclList() altogether.

```
void Program() : {}
{
    DeclList() (Function())* (Main())*
}

void DeclList() : {}
{
    (Decl() <SEMICOLON>)*
}
void Decl() : {}
{
    VarDecl()
    |   ConstantDecl()
}
```

```
void Program() : {}
{
    (Decl())* (Function())* (Main())*
}

void Decl() : {}
{
    VarDecl() <SEMICOLON>
    |   ConstantDecl() <SEMICOLON>
}
```

Kleene Closure makes the code both cleaner and more comprehensible, by reducing its complexity and length.

LEFT FACTORING AND CHOICE CONFLICTS

My initial program was very verbose. I implemented left factoring to 'factor out' prefixes which were common to two or more productions. In NempParameterList() below you can see that I started out with two separate rules. However, it can be seen that the rules share <IDENTIFIER> <COLON> and Type(). Because these are common to both statements I factored them out and created one single rule which covers both individual rules.

```
void NempParameterList() : {}
{
    <IDENTIFIER> <COLON> Type()
    |   <IDENTIFIER> <COLON> Type() <COMMA> NempParameterList()
}
```

```
void NempParameterList() : {}
{
    <IDENTIFIER> <COLON> Type() [<COMMA> NempParameterList()]
}
```

Another example of this is in Fragment(). I did not see this issue until it was flagged when running my program. I was thrown the following error:

> Warning: Choice conflict in (...)* construct at line 216, column 17.
> Expansion nested within construct and expansion following construct
> have common prefixes, one of which is: <IDENTIFIER>
> Consider using a lookahead of 2 or more for nested expansion.

The fix for this was rather simple and involved merging the first three production rules and creating one single rule with <IDENTIFIER> at the head, as it was common to the first two statements. I then added [] around the <LEFTBRACKET> ArgList() <RIGHTBRACKET> to state that it may be used or may be empty as well as adding [<Minus>] as the start to state it may be negative. This then covered all three statements in one line. The problem and subsequent fix can be seen below:

```
void Fragment(): {}
{
    <IDENTIFIER>
    |   <IDENTIFIER> <LEFTBRACKET> ArgList() <RIGHTBRACKET>
    |   <MINUS> <IDENTIFIER>
    |   <NUMBER>
    |   <TRUE>
    |   <FALSE>
}
```

```
void Fragment(): {}
{
    [<MINUS>] <IDENTIFIER> [<LEFTBRACKET> ArgList() <RIGHTBRACKET>]
    |   <NUMBER>
    |   <TRUE>
    |   <FALSE>
}
```

RUNNING EXAMPLES

Once I had the program running and compiling, it was then time to start testing the program with real input.

Looking at the program (Program()) there are three main functions called, namely (Decl())*, (Function())* and Main(). (Decl())* is where all the relevant variables for the program are declared, (Function())* is where all the functions are defined and then Main() is where the program is run. Knowing this was important in being able to come up with effective tests to try and see if there were any bugs.

Firstly, I ran the sample programs which had been provided to us. These programs involved testing case sensitivity, comments, declarations, scopes as well as simple functions and the main function. As well as this I then followed up with some testing of my own.

Here are some of the things I tested to ensure correct functionality:

1. Declaration of multiple variables at the top of the program.
2. Only Boolean operators can be carried out between conditions.
3. Return type of function must be declared.
4. Statements ending with semi-colon.
5. Declaration of multiple functions.
6. Only allowing one main function at the bottom of the program.
7. Order of items in the program is important, Declarations – Functions - Main.
8. Inside each function there can be an arbitrary number of statements, declarations, loops, etc.

9. Functions can pass in any number of variables.
10. Main can contain any number of declarations and statements.