Assignment Submission

Student Name(s):	**Jason Madden**
Student Number(s):	**15486288**
Programme:	**BSc in Computer Applications**
Project Title:	**Compiler Construction - Assignment 1**
Module code:	**CA4009**
Lecturer:	**David Sinclair**
Project Due Date:	**12/11/2018**

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 is a grave and serious offence 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 set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, 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.

I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.

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.library.dcu.ie/citing&refguide08.pdf and/or recommended in the assignment guidelines.

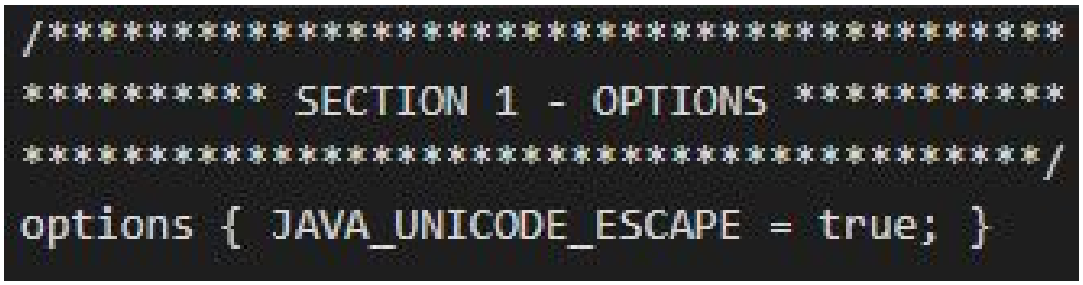
I understand that I may be required to discuss with the module lecturer/s the contents of this submission.

I/me/my incorporates we/us/our in the case of group work, which is signed by all of us.

Signed:Jason Madden

Part one - Options:

Part one of the parser is used to define the options that the parser needs to operate. Since this is simply a lexical analyser the only option needed was to enable unicode escaping. This basically means that our program can accept UTF-8 encoding.



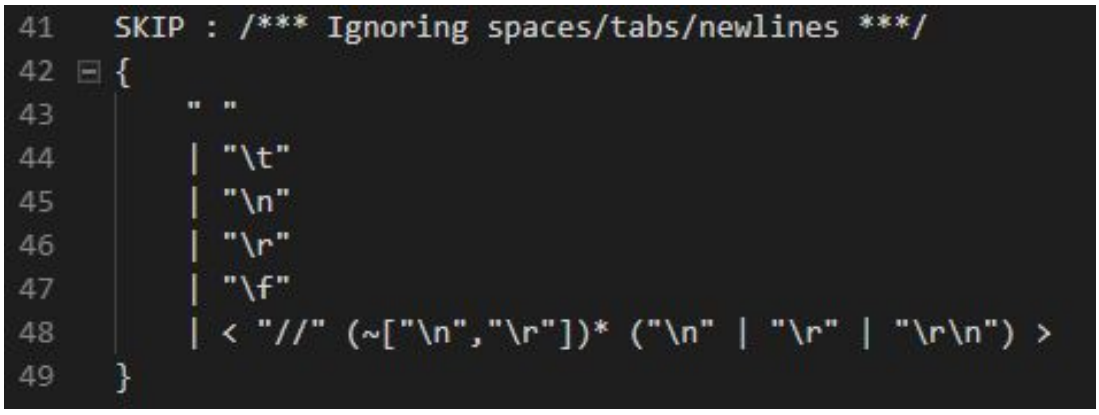
Part two - User Code:

This section is used to start the parsing process. This is where input would be read in from either the command line or from a source file. This is done using a simple if statement. If there is a file given as an argument then the file will be read in as the code to be parsed. If there is no file name given then the input is taken from an input stream from the command line directly from the user. Lastly if there are arguments given but it isn't what is expected to point to a file then there is a message instructing the user how to properly use the parser and point to a source file. Once the source of the code to be parsed is established the input is read in accordingly. If the input stream is from the command line System.in is passed as a parameter when instantiating the SLPParser object whereas when the input stream is from a file java.io.FileInputStream(args[0]) where args[0] is the name of the file. Once the code has been read in to the parser SLPParser.prog() is called as it is the first part of the grammar rules.

```
/*****************************************
******** SECTION 2 - USER CODE ***********
*****************************************/
PARSER_BEGIN(SLPParser)
public class SLPParser {
    public static void main(String args[]) {
        SLPParser parser;
        if (args.length == 0) {
            System.out.println("SLP Parser: Reading from standard input . . .");
            parser = new SLPParser(System.in);
        } else if (args.length == 1) {
            System.out.println("SLP Parser: Reading from file " + args[0] + " . . .");
            try {
                parser = new SLPParser(new java.io.FileInputStream(args[0]));
            } catch (java.io.FileNotFoundException e) {
                System.out.println("SLP Parser: File " + args[0] + " not found.");
                return;
            }
        } else {
            System.out.println("SLP Parser: Usage is one of:");
            System.out.println(" java SLPParser < inputfile");
            System.out.println("OR");
            System.out.println(" java SLPParser inputfile");
            return;
        }
        try {
            parser.Prog(); // start point of the program
            System.out.println("SLP Parser: SLP program parsed successfully.");
        } catch (ParseException e) {
            System.out.println(e.getMessage());
            System.out.println("SLP Parser: Encountered errors during parse.");
        }
    }
}
PARSER_END(SLPParser)
```

Part Three - Token Definition:

Part three describes the tokens that the parser is supposed to recognise. These consist of regular expressions to identify the patterns. Examples of these tokens are <RETURN>, <END> and <BEGIN> which is what the source code will be represented as to make it easier to tell the parser what patterns of tokens are acceptable items of code. The first part of this section is the SKIP sections where you tell the parser to ignore certain patterns or characters such as tabs and newline characters. This can be seen here.



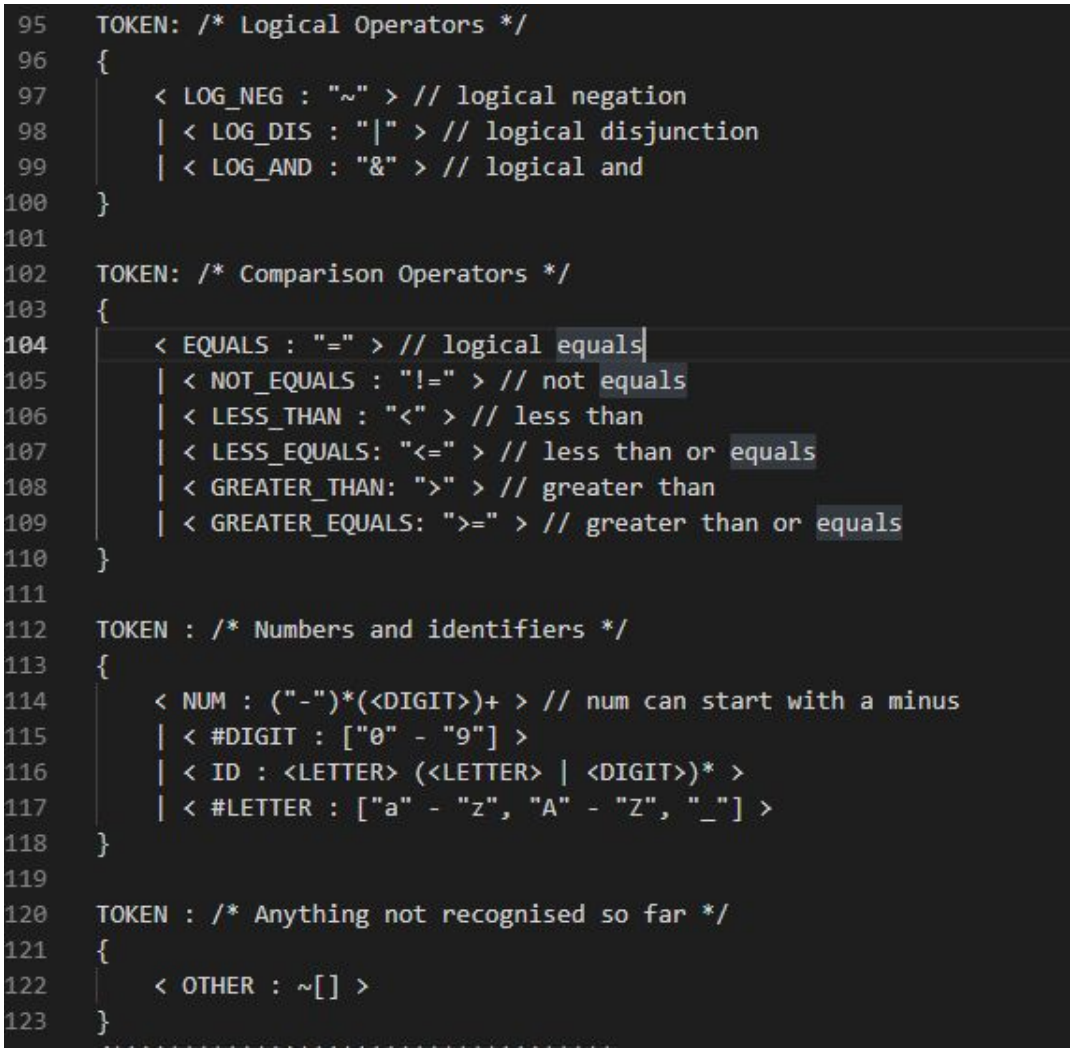
The last line (48) is how the program ignores inline comments. It basically looks for “//” followed by any amount of numbers or letters until it reaches a new line character “\n” or “\r”. Multiline comments are a little trickier as it requires you to be able to keep track of how many open multiline comments there are and make sure there are equal number of closing multiline comments. commentNesting is a number that gets incremented and decremented with every opening and closing multiline comment it encounters. When it reaches 0 again after encountering multiline comments, SwitchTo(DEFAULT) is called. This returns the parser back to the a normal state. If it doesn’t return to DEFAULT an error will be thrown as it was expecting a closing multiline comment.
Next is to define the tokens that the parser is expected to identify in the source code. This makes it easier to define the grammar of the language. This can be seen below.

```
64    TOKEN : /* Keywords and punctuation */
65    {
66        < SEMIC : ";" >
67        | < COLON : ":" >
68        | < ASSIGN : ":=" >
69        | < PRINT : "print" >
70        | < LBR : "(" >
71        | < RBR : ")" >
72        | < COMMA : "," >
73        | < PLUS_SIGN : "+" >
74        | < MINUS_SIGN : "-" >
75        | < DIV_SIGN : "/" >
76    }
```

The next tokens are the components of the language such as if, begin, main and skip.

```
78    TOKEN [IGNORE_CASE]: /* Keywords */
79    {
80        < RETURN : "return" >
81        | < MAIN : "main" >
82        | < BEGIN : "begin" >
83        | < END : "end" >
84        | < VARIABLE : "variable" >
85        | < CONSTANT : "constant" >
86        | < TYPE : ("integer" | "boolean") >
87        | < IS : "is" >
88        | < BOOL : "true" | "false" >
89        | < IF : "if" >
90        | < ELSE : "else" >
91        | < WHILE : "while" >
92        | < SKIP_WORD : "skip" >
93    }
```

These tokens describe the operators of the language such as equals, less than, and and or.



```
95   TOKEN: /* Logical Operators */
96   {
97       < LOG_NEG : "~" > // logical negation
98       | < LOG_DIS : "|" > // logical disjunction
99       | < LOG_AND : "&" > // logical and
100  }
101
102  TOKEN: /* Comparison Operators */
103  {
104      < EQUALS : "=" > // logical equals
105      | < NOT_EQUALS : "!=" > // not equals
106      | < LESS_THAN : "<" > // less than
107      | < LESS_EQUALS: "<=" > // less than or equals
108      | < GREATER_THAN: ">" > // greater than
109      | < GREATER_EQUALS: ">=" > // greater than or equals
110  }
111
112  TOKEN : /* Numbers and identifiers */
113  {
114      < NUM : ("-")*(<DIGIT>)+ > // num can start with a minus
115      | < #DIGIT : ["0" - "9"] >
116      | < ID : <LETTER> (<LETTER> | <DIGIT>)* >
117      | < #LETTER : ["a" - "z", "A" - "Z", "_"] >
118  }
119
120  TOKEN : /* Anything not recognised so far */
121  {
122      < OTHER : ~[] >
123  }
```

The last four lines (120-123) are the catch all at the end that means if there is other characters or words that do not match the token patterns that are defined above, define them as an OTHER token.

Part Four - The Grammar:

The final part of the lexer is where the work happens. So far we have the tokens which makes up the language and the patterns that make up the tokens. But until now the tokens don't mean much. That is where the grammar comes in. The grammar is what tells the lexer what sequence of tokens are legal in the CAL language. The code starts by saying the parser is expecting a massive file of code followed by an EOF. code breaks down into a block and more code and a block of code is made up of either a decl_list(), func_list() or a main_statement(). This recursion is what breaks the code up into manageable chunks. This can be seen below.

```
124  /************************************
125  ***** SECTION 4 - THE GRAMMAR *****
126  ************************************/
127
128  void Prog() : {}
129  {
130      code() <EOF>
131  }
132
133  void code() : {}
134  {
135      (block() [code()])
136  }
137
138  void block() : {}
139  {
140      decl_list()
141      | func_list()
142      | main_statement()
143  }
144
```

The same pattern can be seen when defining decl_list() as a decl_list() consists of decl() followed by the rest of the list. decl() can either be var_decl() or const_decl() which both break down into patterns to represent either a constant declaration or a variable declaration as seen below.

```
145  void decl_list() : {}
146  {
147      (decl() [<SEMIC> decl_list()])
148  }
149
150  void decl() : {}
151  {
152      var_decl()
153      | const_decl()
154  }
155
156  void var_decl() : {}
157  {
158      (<VARIABLE> [<ID> <COLON> <TYPE> <SEMIC>])
159  }
160
161  void const_decl() : {}
162  {
163      (<CONSTANT> [<ID> <COLON> <TYPE> <ASSIGN> expr() <SEMIC>])
164  }
165
```

For defining the func_list() yet again it is simply a func() followed by the rest of the func_list(). func() then breaks down to a sequence of tokens that represent a function.

```
166  void func_list() : {}
167  {
168      (func() [func_list()])
169  }
170
171  void func() : {}
172  {
173      <TYPE> <ID> <LBR> (param_list())* <RBR> <IS> (decl_list())* (statement_block())
174  }
```

As seen in the function definition, there is a statement block that can be in a function. A statement is the main parts of the function. A statement can be if(x < 10) {i = i + 1;} for example. Statements also include else statements, while loops and skip. The definition can be seen below.

The main statement is very similar to the function definition where it is a statement block wrapped around MAIN BEGIN and ending with an END token. This can be seen below.

```
272  void main_statement() : {}
273  {
274      <MAIN> <BEGIN> (decl_list())* (statement_block())* <END>
275  }
```

This is the main components of my lexical analyser written in javacc.

```
176  void statement_block() : {}
177  {
178      (statement() [statement_block()])
179  }
180
181  void statement() : {}
182  {
183      LOOKAHEAD(2) (<ID> <ASSIGN> expr() <SEMIC>)
184      | LOOKAHEAD(2) <ID> <LBR> (expr())* <RBR> <SEMIC>
185      | <BEGIN> (statement_block())* return_statement() <END>
186      | <IF> condition() <BEGIN> (statement_block())* <END>
187      | <ELSE> <BEGIN> (statement_block()) * <END>
188      | <WHILE> condition() <BEGIN> (statement_block())* <END>
189      | <SKIP_WORD> <SEMIC>
190  }
```

In the statement definition, expr() is used which is an expression definition. Expressions can be 4 - 4, i + j and func(x, y, c) and the definition for it can be seen below.

```
220  void expr() : {}
221  {
222      LOOKAHEAD(3) fragment() bin_op() fragment()
223      | LOOKAHEAD(2) (<ID>)* <LBR> expr() <RBR>
224      | arg_list()
225      | fragment()
226  }
227
228  void arg_list() : {}
229  {
230      nemp_arg_list()
231  }
232
233  void nemp_arg_list() : {}
234  {
235      (<ID> (<COMMA> nemp_arg_list())*)
236  }
237
238  void fragment() : {}
239  {
240      <ID>
241      | <MINUS_SIGN> <ID>
242      | <NUM>
243      | <BOOL>
244  }
```

As before the arg_list is defined with the same recursive pattern that was seen in the func_list and the decl_list by telling the lexer to expect an ID COMMA followed by the rest of the list. The screenshot also shows the definition of a fragment which a lot of the

components of expressions are composed of. bin_op() that can be seen simply breaks down to plus or minus tokens.