Assignment 1 A Lexical and Syntax Analyser for the CCAL Language

I first began writing the program by setting the options for the language, I set the program to ignore case as the CCAL language is not case sensitive.

```
/* S1 - OPTIONS */

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

Then I proceeded to writing the java compilation unit for the program, this is in the "S2-USER CODE" section. I used the template provided in the JAVACC notes for SLPParser as the basis of this section changing it as appropriate for the CCAL parser. This section sets the program to either read from standard input or read from a file e.g "java Ccalparser test.txt". Upon completing the parsing, the program will output either "Program parsed!" or "There was a problem parsing this program".

The next step was writing the Token definitions this is found in the "S3 – TOKEN DEFINITIONS" section. I began by writing the SKIP method, I used the method in the JAVACC notes as the basis here. This dealt with tabs, spaces and newlines and ignored them as they arose. The SKIP method also skipped one-line comments (e.g "// comment") as well as multi-line comments and nested comments (e.g "/* comment /* nested comment */*/) this worked by counting the amount of "/*" the program took and matched it with the amount of "*/".

```
SKIP : //COMMENTS
{

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


<IN_COMMENT> SKIP : // Dealing wiht nested comments
{
  "/*" { comment_nest++; }
| "*/" { comment_nest--;
              if (comment_nest == 0)
              SwitchTo(DEFAULT);}
| <~[]>
}
```

I then went on to create the TOKEN method using the one in the JAVACC notes as a guide, I first defined the keywords set out by the CCAL language then all the punctuation and symbols and finally the numbers and identifiers as shown below. The CCAL language accepts numbers 0 – 9 that can be negative or positive it accepts 0 but may not have a number starting with a 0, the identifiers are represented by letter followed by letters, digits or an underscore. I took extra care in making sure these were appropriately written.

```
TOKEN : //keywords

{
 <VAR : "var">
|<INT : "integer">
|<CONST : "const">
|<RET : "return">
|<BOOL : "boolean">
|<VOID : "void">
|<MAIN : "main">
|<IF : "if">
|<ELSE : "else">
|<TRUE : "true">
|<FALSE : "false">
|<WHILE : "while">
|<SKIP_ : "skip">
}
```

```
TOKEN : //punctuation and symbols
{
 <COMMA : ",">
|<ASSIGNMENT: "=">
|<DOT : ".">
|<SEMI_COLON : ";">
|<COLON : ":">
|<LEFT_BRACE : "{">
|<RIGHT_BRACE : "}">
|<LEFT_BRACKET : "(">
|<RIGHT_BRACKET : ")">
|<PLUS : "+">
|<MINUS : "-">
|<NOT : "~">
|<OR : "||">
|<AND : "&&">
|<EQUALTO : "==">
|<NOT_EQUAL : "!=">
|<LESS_THAN : "<">
|<GREATER_THAN : ">">
|<LTHAN_EQUALTO : "<=">
|<GTHAN_EQUALTO : ">=">
}
```

```
TOKEN : //NUMBERS AND ID

{
  <NUMBER : "0"|(<MINUS>)?["1"-"9"](<DIGIT>)*>
| <#DIGIT : ["0" - "9"]>
| <ID: <LETTER>(<LETTER>|<DIGIT>|"_")*>
| <#LETTER : ["a" - "z", "A" - "Z"]>
}

TOKEN : // Anything not recognised so far
{
<OTHER : ~[]>
}
```

After all the rules for the language were set up, I began writing the grammar as set out in the CCAL language. This is found in the “S4 – THE GRAMMAR” section. I used the JAVACC notes to understand how to write the grammar rules. After they were all written out the first run of the program showed me a warning that there were two rules with a left recursion error. One in the fragment ("fragment -->expression-->fragment”) production rule and one in the condition production rule ("condition--> condition"). To remove it I used the notes and “cs.lmu.edu/~ray/notes/javacc/” for help. For condition() I created a prime condp() so that the alpha and beta were clearer and to remove the left recursion. For expression I followed similar steps and created “pr()” this also removed ambiguity and a choice conflict as a result.

```
//created condp() to deal with left recursion
void condition():{}{
  <NOT> condition()
  | <LEFT_BRACKET> condition() <RIGHT_BRACKET> condp()
  | fragment() comp_op() expression() condp()
}

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

//conflict resolved with pr func
void fragment(): {} {
  <MINUS> <ID> | <ID> (<LEFT_BRACKET> arg_list() <RIGHT_BRACKET>)? | <NUMBER> | <TRUE> | <FALSE>
}

//created to resolve conflict
void pr(): {} {
  (binary_arith_op() fragment() pr())?
}

//conflict resolved wiht pr func
void expression():{}{
  fragment() pr() | <LEFT_BRACKET> expression() <RIGHT_BRACKET> pr()
}
```

After the left recursion issues were resolved I ran the program again and this presented me with 5 choice conflict warnings. I worked through each removing ambiguity and this resolved all the warnings and allowed me to successfully run the program. The final step was to test it by passing the test files provided in the assignment handout. Once all these parsed, I was happy the parser was complete.

How to Run:

1. In command line enter “javacc ccal.ccl”
2. Then “javac *.java” in the same folder as above
3. Then “java CcalParser” to start reading from standard input
4. Or “java CcalParser test.txt” to read form a text file