Name	Jordan Healy
Student Number	13379226
Programme	CASE
Module Code	CA4003
Assignment Title	A Lexical and Syntax Analyser for the CCAL Language
Submission Date	Monday 7th November 2016

1 CONTENTS

2 INTRODUCTION

In this report I will outline my thought process for creating a lexical and syntax analyser for the CCAL language.

Taking the template for a javacc file described in the Lexical Analyser notes, we have four sections.

1. Options
2. User Code
3. Token Definitions
4. Grammar & Production Rules

I created seven input files that contain the contents of each example from the CCAL.pdf file. They will be referred to from now on as inputfile01.ccl, inputfile02.ccl, inputfile03.ccl etc.

3 OPTIONS

Our parser will have only one option.

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

This is because the language is not case sensitive.

4 USER CODE

All javacc parsers must begin with a declaration with its parser name, in this case it is.

```
PARSER_BEGIN(Assignment1)
```

Our user code will initialise the parser if the input file of the language code is passed as a command line argument. If not, the system exits with an instruction to the user on how to run the parser.

```
Assignment1 tokeniser;
// Initialise parser
if(args.length == 1) {
    try {
        tokeniser = new Assignment1(new FileInputStream(args[0]));
    }
    catch(FileNotFoundException e) {
        System.err.println("File " + args[0] + " not found");
        return;
    }
} else {
    System.out.println("Assignment1 can be used by entering the following
command:");
    System.out.println("     java Assignment1 inputfile");
    return;
}
```

We also want to display the contents of the file by each token and its corresponding lexeme. To make this loop of code easier to read, I've put each tokens kind number and its name into a HashMap for easier access.

```
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");
allTokens.put(ELSE, "ELSE");
allTokens.put(WHILE, "WHILE");
allTokens.put(SKIP_W, "SKIP_W");
allTokens.put(TRUE, "TRUE");
allTokens.put(COMMA, "COMMA");
allTokens.put(SEMICOLON, "SEMICOLON");
allTokens.put(COLON, "COLON");
allTokens.put(LEFT_BRACE, "LEFT_BRACE");
allTokens.put(RIGHT_BRACE, "RIGHT_BRACE");
allTokens.put(LEFT_BRACKET, "LEFT_BRACKET");
allTokens.put(RIGHT_BRACKET, "RIGHT_BRACKET");
allTokens.put(ASSIGNMENT, "ASSIGNMENT");
allTokens.put(PLUS, "PLUS");
allTokens.put(MINUS, "MINUS");
allTokens.put(NOT, "NOT");
allTokens.put(OR, "OR");
```

```
allTokens.put(AND, "AND");
allTokens.put(EQUALS, "EQUALS");
allTokens.put(NOT_EQUALS, "NOT_EQUALS");
allTokens.put(LESS_THAN, "LESS_THAN");
allTokens.put(LESS_THAN_OR_EQUAL_TO, "LESS_THAN_OR_EQUAL_TO");
allTokens.put(GREATER_THAN, "GREATER_THAN");
allTokens.put(GREATER_THAN_OR_EQUAL_TO, "GREATER_THAN_OR_EQUAL_TO");
allTokens.put(NUMBER, "NUMBER");
allTokens.put(IDENTIFIER, "IDENTIFIER");

// Display file
System.out.println("For the following file...");
for(Token t = getNextToken(); t.kind!=EOF; t = getNextToken()) {
    System.out.print(allTokens.get(t.kind) + " ");
    System.out.print("('" + t.image + "') ");
}
```

Then we call `tokeniser.program();` to parse the program. This is wrapped in a try catch block for a ParseException, and if an error is caught then the program is not valid CCAL code. But because of we've displayed the file tokens previously the parser will try to read tokens from the end of file, therefore we must re-initialise the parser.

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

And of course our parser must be closed off with its declaration.

```
PARSER_END(Assignment1)
```

5 TOKEN DEFINITIONS

Firstly, all token definitions were written as described in the CCAL.pdf file. Each type of token is separated and indicated by comment title.

```
/* Reserved Words */
/* Symbols */
/* Operators */ . . .
```

When compiling my parser, some ParseException errors were generated regarding my token definitions.

```
org.javacc.parser.ParseException: Encountered " "SKIP" "SKIP "" at line 82,
column 7.
```

The definition for the token <SKIP> was clashing with the regular expression for skipping characters in this language. The fix was to rename this token to <SKIP_W>, shorthand for <SKIP_WORD>.

I would also get an error for matching string literals.

```
"\t" cannot be matched as a string literal token
"\n" cannot be matched as a string literal token
"\r" cannot be matched as a string literal token
"\f" cannot be matched as a string literal token
" " cannot be matched as a string literal token
```

I had a token definition < OTHER : ~[] > that would be anything not recognised by any of the other specified tokens. The fix was to move this regular expression into the SKIP regular expressions.

Once my javacc file had compiled and created all the necessary java files, I attempted to compile all of the .java files. An error occurred when trying to compile Assignment1TokenManager.java.

```
Assignment1TokenManager.java:807: error: cannot find symbol
            commentNesting++;
            ^
            symbol:   variable commentNesting
            location: class Assignment1TokenManager
```

This was because I had forgotten to declare commentNesting for my regular expression for skipping nested comments. So I added the following.

```
TOKEN_MGR_DECLS :
{
    static int commentNesting = 0;
}
```

When running the parser with inputfile04.ccl a lexical error was generated.

```
Exception in thread "main" TokenMgrError: Lexical error at line 3, column
7.  Encountered: " " (32), after : "//"
        at
Assignment1TokenManager.getNextToken(Assignment1TokenManager.java:798)
        at Assignment1.getNextToken(Assignment1.java:837)
```

```
        at Assignment1.main(Assignment1.java:63)
```

My `SKIP` regular expression couldn't find a space character in single line comments, so I changed the regular expression so that it deals with all ASCII characters from SPACE to TILDA.

```
SKIP : {
    < "//" (~["\n","\r"])* ("\n" | "\r" | "\r\n") >
  |  . . .
}
```

Another lexical error was generated when running the parser with inputfile07.ccl.

```
Exception in thread "main" TokenMgrError: Lexical error at line 8, column
11.  Encountered: "0" (48), after : ""
        at
Assignment1TokenManager.getNextToken(Assignment1TokenManager.java:798)
        at Assignment1.getNextToken(Assignment1.java:877)
        at Assignment1.main(Assignment1.java:62)
```

My regular expression for `NUMBER` was incorrect. `NUMBER`'s cannot start with a 0 but can be negative. I left out the case where `NUMBER`'s can start with 0 if and only if there are no other `DIGITS` following it (therefore the number is 0).

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

6 GRAMMAR & PRODUCTION RULES

The production rules were written as per the CCAL.pdf file, then were later changed and transformed according to errors found when compiling and running the parser. All epsilon symbols found in production rules in CCAL.pdf are replaced by the null set which is denoted by `{}`.

6.1 KLEENE CLOSURE

I learned that you can use Kleene closure when defining production rules, because of this there are some rules I can transform. Namely, the rules `nemp_parameter_list()` `nemp_arg_list()`.

`nemp_parameter_list()` is used to list one or more identifiers as parameters for declaring a function. `nemp_parameter_list()` calls itself so that it can call
`<IDENTIFIER> <COLON> type()`
again. So we can change this rule to
`<IDENTIFIER> <COLON> type() ( <COMMA> <IDENTIFIER> <COLON> type() )*`
and rename it `parameter()`. So any where we see `parameter_list()` we can replace it with `( parameter() )*`.
So either `number ; integer` or `number ; integer , isMale ; boolean` are both valid CCAL language code.

```
void parameter ( ) : {}
{
    <IDENTIFIER> <COLON> type() ( <COMMA> <IDENTIFIER> <COLON> type() )*
}
```

The same can be done to `nemp_arg_list()`.

```
void arg ( ) : {}
{
    <IDENTIFIER> ( <COMMA> <IDENTIFIER> )*
}
```

I discovered that `decl_list()` and `decl()` can be transformed in a similar vein, to remove `decl_list()` altogether. `decl_list()` just adds a `<SEMICOLON>` to any number of `decl()` calls, so `<SEMICOLON>` can be put to the end of `decl()` and replace any call of `decl_list()` to `( decl() )*`.

This can also be done to `function_list()` and `function()`.

6.2 LEFT-RECURSION

We can also remove some left-recursion by using Kleene closure.

```
Error: Line 218, Column 1: Left recursion detected: "statement_block... -->
statement_block..."
```

The function `statement_block()` is no longer needed as it can be replaced with the Kleene closure of `statement()`.

There was also left-recursion for the `expression()` production rule.

```
Error: Line 240, Column 1: Left recursion detected: "expression... -->
fragment... --> expression..."
```

The fix was to add <LEFT_BRACKET> and <RIGHT_BRACKET> tokens to the expression() call in the fragment() function.

```
void fragment ( ) : {}
{
    ( ...
      <LEFT_BRACKET> expression() <RIGHT_BRACKET> |
      ... )
}
```

There was also left-recursion for the condition() production rule.

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

The comp_op() function is only called in condition(), therefore I can manipulate this production rule without it affecting any other rules. I decided to put the <OR> | <AND> tokens into the comp_op() function to reuse this rule.
For the rule
condition() ::= condition() comp_op() condition()
we can move the final condition() call into the comp_op() rule.

```
void comp_op ( ) : {}
{
    ( <EQUALS> condition() |
      <NOT_EQUALS> condition() |
      <LESS_THAN> condition() |
      <LESS_THAN_OR_EQUAL_TO> condition() |
      <GREATER_THAN> condition() |
      <GREATER_THAN_OR_EQUAL_TO> condition() |
      <OR> condition() |
      <AND> condition() |
      {} )
}
```

Similar to this, the last expression() call in
condition() ::= expression() comp_op() expression()
can be removed as comp_op() will call condition() which in turn will call expression() comp_op().

Finally, merge rules condition() ::= <LEFT_BRACKET> condition() <RIGHT_BRACKET> and condition() ::= condition() comp_op() together and add epsilon to comp_op()

```
void condition ( ) : {}
{
    ( ...
      <LEFT_BRACKET> condition() <RIGHT_BRACKET> comp_op() |
      ... )
}
void comp_op ( ) : {}
{
```

```
    ( ...
      {} )
}
```

6.3 Other Transformations

I noticed that my `function()` and `expression()` rules could be transformed. The part of the rule where `expression()` is called doesn't need the `<LEFT_BRACKET>` `<RIGHT_BRACKET>` tokens as they will be called in `expression()`. `<LEFT_BRACKET>` `expression()` `<RIGHT_BRACKET>` is called in `fragment()`, which is called in `expression()` so therefore this can be taken out `fragment().expression() ::= fragment() binary_arith_op() fragment()` can be transformed similar to `condition()` that was described previously.

```
void expression ( ) : {}
{
    ( ...
      fragment() binary_arith_op() )
}
void binary_arith_op ( ) : {}
{
    ( <PLUS> expression() |
      <MINUS> expression() |
      {} )
}
```

6.4 Choice Conflicts

There were five choice conflicts, two of which were solved by transforming `nemp_parameter_list()` and `nemp_arg_list()` using Kleene closure as stated above. Unfortunately, there were three choice conflicts with `statement()`, `expression()` and `condition()`. Each of which I used lookahead of 2.

7 PARSING EXAMPLE CCAL CODE

I attempted to parse each of the examples from the CCAL.pdf file. The first three files parsed correctly, however the other files did give lexical errors or parse exceptions.

7.1 INPUTFILE04.CCL

```
Exception in thread "main" TokenMgrError: Lexical error at line 4, column
6.  Encountered: "\u2217" (8727), after : "/"
        at
Assignment1TokenManager.getNextToken(Assignment1TokenManager.java:798)
        at Assignment1.getNextToken(Assignment1.java:837)
        at Assignment1.main(Assignment1.java:62)
```

When I copied the text from CCAL.pdf into inputfile04.ccl there were '∗' (registered as \u2217) characters instead of '*' characters. This was a copy and paste mistake made by myself, so I simply changed those characters in the input file.

7.2 INPUTFILE05.CCL

7.2.1 Error 1

```
ParseException: Encountered " ";" "; "" at line 3, column 14.
      Was expecting:
          "}" ...
```

The parser is parsing two consecutive semicolons. So its attempting to parse `return ();;` instead of `return ();`.

The `function()` rule is the only production rule using the `<RETURN>` token, so the problem must be there. The issue was with `<RETURN> ( expression() | {} ) <SEMICOLON>`, as when the parser would find a `<LEFT_BRACKET>` it would call `fragment()` (from `expression()`) looking for a `<RIGHT_BRACKET>` but not finding one. So I added epsilon to `fragment()` and removed it from `<RETURN> ( expression() | {} ) <SEMICOLON>`.

7.2.2 Error 2

```
ParseException: Encountered " ";" "; "" at line 8, column 12.
      Was expecting one of:
            "if" ...
            "while" ...
            "skip" ...
            "{" ...
            "}" ...
            <IDENTIFIER> ...
```

The parser is not expecting a `<SEMICOLON>` on the line `func ();`. The fix for this was to add `<SEMICOLON>` to the rule `statment() ::= call_args()` to create `statment() ::= call_args() <SEMICOLON>`.

7.3 INPUTFILE06.CCL

```
ParseException: Encountered " <NUMBER> "2 "" at line 7, column 9.
```

```
      Was expecting one of:
          "(" ...
          <IDENTIFIER> ...
```

This was fixed from the issue in inputfile05.ccl

7.4 INPUTFILE.07.CCL

7.4.1 Error 1

```
Exception in thread "main" TokenMgrError: Lexical error at line 11, column
9.  Encountered: "\u2212" (8722), after : ""
        at
Assignment1TokenManager.getNextToken(Assignment1TokenManager.java:807)
        at Assignment1.getNextToken(Assignment1.java:877)
        at Assignment1.main(Assignment1.java:62)
```

The line `x = -x;` was invalid due to the **<**MINUS**>**. The fix was similar to the one in inputfile04.ccl as the minus character was a '–' (registered as \u2212) instead of the standard '-' charachter. So I changed the characters in the input file.

7.4.2 Error 2

```
ParseException: Encountered " "if" "if "" at line 13, column 8.
      Was expecting:
           "{" ...
```

The production rule

```
statement() ::= <IF> condition()
                    <LEFT_BRACE> ( statement() <SEMICOLON> )* <RIGHT_BRACE>
                <ELSE>
                    <LEFT_BRACE> ( statement() <SEMICOLON> )* <RIGHT_BRACE>
```

can be changed to

```
statement() ::= <IF> condition()
                    statement()
                <ELSE>
                    statement()
```

since

```
statement() ::= <LEFT_BRACE> ( statement() <SEMICOLON> )* <RIGHT_BRACE>
```

7.4.3 Error 3

```
ParseException: Encountered " <IDENTIFIER> "result "" at line 28, column 3.
      Was expecting:
           ";" ...
```

This was to do with the **<**SEMICOLON**>** in `statement()`. Any call to

```
( statement() <SEMICOLON> )*
```

is replaced by

```
( statement() )*
```

so that the **<**SEMICOLON**>** is called by

```
<IDENTIFIER> <ASSIGNMENT> expression() <SEMICOLON>.
```

7.4.4 Error 4

```
ParseException: Encountered " "skip" "skip "" at line 40, column 5.
      Was expecting one of:
           "if" ...
           "while" ...
           "{" ...
           "}" ...
           <IDENTIFIER> ...
```

I forgot to add the production rule

```
statement() ::= <SKIP_W> <SEMICOLON>.
```

8 HOW TO COMPILE AND RUN MY PARSER

Do the following commands to compile the parser

```
javacc Assignment1.jj
javac *.java
```

The parser can be run as so

```
java Assignment1 inputfile
```