Declaration on Plagiarism

Assignment Submission Form

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

Name(s): Ore Ibikunle
Programme: Computer Applications
Module Code: CA4003
Assignment Title: Lexical and Syntax Analyser
Submission Date: 04/11/2018
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 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/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): _________Ore Ibikunle____________ Date: ______03/11/2018______

Table of Contents

Lexical and Syntax Analyser

1. Introduction

The goal of this assignment was to implement a lexical and syntax analyser using JavaCC for a language known as CAL.

The purpose of lexical analysis is to convert a stream of characters from the source program into a stream of tokens that represent recognised keywords, identifiers, numbers, and punctuation. It consists of defining lexical rules that define the alphabet of the language and how these characters are combined to form valid words.

The goal of syntax analysis is to combine the tokens generated by the lexical analysis into a valid sentence. This involves defining syntax rules that define how valid words are combined to form valid statements.

This report will outline the step by step process of creating a lexical and syntax analyser for the CAL language.

2. Design

As described by the notes, the JavaCC program will consist of four sections:

1. Options
2. User Code
3. Tokens
4. Production Rules

2.1 Options:

The parser will have two options:

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

The CAL language is not case sensitive, so setting **IGNORE_CASE** to true will cause the token manager to ignore case in the token specifications and the input files.

2.2 User Code:

After we have specified the options we then specify the user code. In JavaCC, all parsers must begin with a declaration with its parser name.

```
PARSER_BEGIN(Cal)
```

As specified by the CAL documentation, source code should be kept in files with the **.cal** extension, so input files will be passed as a command line argument. If there is no command line argument, then the system exits the program with an output message.

```
Cal parser;
if(args.length == 0)
{
    System.out.println("Enter a file with extension .cal");
    System.exit(1);
}
else if(args.length == 1)
{
    try
    {
        parser = new Cal(new FileInputStream(args[0]));
        parser.Program();
        System.out.println("The program was parsed successfully.");
    }
    catch(FileNotFoundException e)
    {
        System.out.println("File " + args[0] + " doesn't exist.");
    }
    catch(ParseException e)
    {
        System.out.println("The program did not parse.");
        System.out.println(e.getMessage());
    }
}
else
{
    System.out.print("Type: java Cal YOUR_FILE_NAME.cal");
}
```

The user code will handle two exceptions that could be thrown by our program. A **FileNotFoundException** will be thrown if the file passed as a command line argument does not exist. A **ParseException** will be thrown if the syntax specified by the user is invalid and does not match the grammar specified by the CAL language. We call **parser.Program()** to parse our program.

After defining the user code, all JavaCC parsers must end with a declaration with its parser name.

```
PARSER_END(Cal)
```

2.3 Tokens:

All tokens in the program are as specified by the CAL documentation. Firstly, we define what our parser should skip, which are spaces, newlines, returns, and form feeds.

```
SKIP : /*** Ignoring spaces/tabs/newlines ***/
{
      " "
    | "\t"
    | "\n"
    | "\r"
    | "\f"
}
```

As specified by the CAL documentation, there are two forms of comment, one is delimited by **/*** and ***/** and can be nested; the other begins with **//** and is delimited by the end of line and this type of comment cannot be nested.

The following code from our notes helps us to handle nested comments.

```
SKIP : /* COMMENTS */
{
    "/*" { commentNesting++; } : IN_COMMENT
}

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

To handle comments delimited by **//**, I used this code:

```
SKIP : /* Single comments */
{
    < "//" (~["\n"])* "\n" >
}
```

This will match with **//** followed by anything that is not a newline character up until it reaches a newline character.

Secondly, I defined the reserved words and operators which were specified by the CAL language documentation.

```
TOKEN: /* keywords */
{
    < VARIABLE : "variable">
|   < CONSTANT : "constant">
|   < RETURN : "return">
|   < INTEGER : "integer">
|   < BOOLEAN: "boolean" >
|   < VOID: "void" >
|   < MAIN: "main" >
|   < IF: "if">
|   < ELSE: "else" >
|   < TRUE: "true" >
|   < FALSE: "false" >
|   < WHILE: "while" >
|   < BEGIN: "begin" >
|   < END: "end" >
|   < IS: "is" >
|   < SKP: "skip" >
}

TOKEN: /* operators */
{
    < COMMA : ",">
|   < SEMIC: ";" >
|   < COLON: ":" >
|   < ASSIGNMENT: ":=" >
|   < LP: "(" >
|   < RP: ")" >
|   < PLUS: "+" >
|   < MINUS: "-" >
|   < NOT_OP: "~" >
|   < OR: "|" >
|   < AND: "&" >
|   < EQUAL_TO: "=" >
|   < NEQUAL_TO: "!=" >
|   < LESS_THAN: "<" >
|   < LEQUAL_TO: "<=" >
|   < GREATER_THAN: ">" >
|   < GEQUAL_TO: ">=" >
}
```

I then specified the tokens for our integers and identifiers. From the CAL documentation is states that an **identifier** cannot be a reserved word and is represented by a string of letters, digits, or underscore (“_”) beginning with a letter. Meanwhile, an **integer** is represented by a string of one or more digits (**“0”-“9”**) and may start with a minus sign (“**-**”).

```
TOKEN: /* Numbers and Identifiers */
{
    < #DIGIT: ["0"-"9"] >
|   < #LETTER: ["a"-"z", "A"-"Z"] >
|   < NUMBER: ("-")? "0" | ("-")? ["1"-"9"] (< DIGIT >)* >
|   < IDENTIFIER: < LETTER > (< LETTER > | < DIGIT > | "_")* >
}
```


```
TOKEN: /* Numbers and Identifiers */
{
    < #DIGIT: ["0"-"9"] >
|   < #LETTER: ["a"-"z", "A"-"Z"] >
|   < NUMBER: ("-")? "0" | ("-")? ["1"-"9"] (< DIGIT >)* >
|   < IDENTIFIER: < LETTER > (< LETTER > | < DIGIT > | "_")* >
}
```


For the CAL language numbers may not start with leading “**0**”s, e.g. the number **0012** is illegal. A zero may also start with a minus sign.

Lastly, I created a token to recognize any other character that hasn’t been specified by our other tokens.

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

2.4 Production Rules:

Some of the production rules specified by the CAL language are ambiguous (meaning they can have more than one leftmost derivation) and contain left recursion.

Since JavaCC is an **LL(1)** parser (meaning it does a left to right scan, left most-derivation, with a 1-token lookahead) the production rules cannot contain left recursion and I will have to remove left recursion by using left factoring.

By default, JavaCC will decide which branch to take by looking at the next token. There is no **backtracking** in JavaCC. If the token chosen is compatible with the first choice, the first choice is taken, and that decision is irreparable. The choices the parser must make will cause choice conflicts. These can be removed by either using left factoring, factoring out a common prefix, or by using a syntactic **LOOKAHEAD**.

Removing choice conflicts by using left factoring is better than using a syntactic **LOOKAHEAD**, but it makes the code harder to manage for the second assignment. Using a **LOOKAHEAD** allows for the code to be easily maintained and optimised for the second assignment when we start to build the **Abstract Syntax Tree** (**AST**) for our language. The only knockoff is that it reduces the quality of the code. So, when possible I will factor out a common prefix to remove choice conflicts from the production rules rather than use left factoring or a syntactic **LOOKAHEAD**.

3. Implementation

3.1 Choice Conflicts:

In this section, I will further discuss how I removed choice conflicts from the production rules that were ambiguous.

The first choice conflict that I had to handle was for the **Nemp_parameter_list()** production rule. Which I removed by using a syntactic **LOOKAHEAD**. But, as I mentioned above, using a syntactic **LOOKAHEAD** reduces the quality of the code.

```
void Nemp_parameter_list() : { }
{
    LOOKAHEAD(3) < IDENTIFIER > < COLON > Type()
|   < IDENTIFIER > < COLON > Type() < COMMA > Nemp_parameter_list()
}
```

When you look at the **Nemp_parameter_list()** production rule in the CAL documentation:

$$\begin{aligned} \langle \text{parameter\_list} \rangle &\models \langle \text{nemp\_parameter\_list} \rangle \mid \epsilon && (9) \\ \langle \text{nemp\_parameter\_list} \rangle &\models \text{identifier:} \langle \text{type} \rangle \mid \text{identifier:} \langle \text{type} \rangle \text{, } \langle \text{nemp\_parameter\_list} \rangle \end{aligned}$$

There is a choice conflict here because this production rule is ambiguous. So, after I factored out the common prefix which was **identifier:<type>**, this allowed me to remove the **LOOKAHEAD** I used previously.

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

The use of square brackets around **[< COMMA > Nemp_parameter_list()]** make it optional.

The next choice conflict that I had to remove was in the **Statement()** production rule. Once again for demonstration purposes, I used syntactic **LOOKAHEAD** to remove this choice conflict:

```
void Statement() : {}
{
    LOOKAHEAD(2)< IDENTIFIER > < ASSIGNMENT > Expression() < SEMIC >
|   < IDENTIFIER > < LP > Arg_list() < RP > < SEMIC >
|   < BEGIN > Statement_block() < END >
|   < IF > Condition() < BEGIN > Statement_block() < END >
    < ELSE > < BEGIN > Statement_block() < END >
|   < WHILE > Condition() < BEGIN > Statement_block() < END >
|   < SKP > < SEMIC >
}
```

When you look at the **Statement()** production rule:

$$
\begin{aligned}
\langle \text{statement} \rangle \quad \models \quad & \text{identifier} := \langle \text{expression} \rangle \,; \;\mid \\
& \text{identifier} \,(\, \langle \text{arg\_list} \rangle \,)\,; \;\mid \\
& \textbf{begin} \ \langle \text{statement\_block} \rangle \ \textbf{end} \;\mid \\
& \textbf{if} \ \langle \text{condition} \rangle \ \textbf{begin} \ \langle \text{statement\_block} \rangle \ \textbf{end} \\
& \textbf{else begin} \ \langle \text{statement\_block} \rangle \ \textbf{end} \;\mid \\
& \textbf{while} \ \langle \text{condition} \rangle \ \textbf{begin} \ \langle \text{statement\_block} \rangle \ \textbf{end} \;\mid \\
& \textbf{skip} \;;
\end{aligned} \tag{12}
$$

Everything between **identifier** and semicolon (“**;**”) is optional, meaning that we can factor it out using **OR**.

```
void Statement() : {}
{
    < IDENTIFIER > (< ASSIGNMENT > Expression() | < LP > Arg_list() < RP >) < SEMIC >
|   < BEGIN > Statement_block() < END >
|   < IF > Condition() < BEGIN > Statement_block() < END >
    < ELSE > < BEGIN > Statement_block() < END >
|   < WHILE > Condition() < BEGIN > Statement_block() < END >
|   < SKP > < SEMIC >
}
```

This is because between **identifier** and semicolon (“**;**”) there can either be **<ASSIGNMENT> Expression()** OR **<LP> Arg_list() <RP>**. Unlike when I used the square brackets, one of these **must** occur.

Finally, the last choice conflict I had to handle was for the **Nemp_arg_list()**. Once again to demonstrate, I used a **LOOKAHEAD** to remove this choice conflict:

```
void Nemp_arg_list() : { }
{
    LOOKAHEAD(2) < IDENTIFIER >
|   < IDENTIFIER > < COMMA > Nemp_arg_list()
}
```

When we look at the **Nemp_arg_list()** production rule:

$$\langle \text{arg\_list} \rangle \models \langle \text{nemp\_arg\_list} \rangle \mid \epsilon \quad (18)$$
$$\langle \text{nemp\_arg\_list} \rangle \models \text{identifier} \mid \text{identifier}, \langle \text{nemp\_arg\_list} \rangle \quad (19)$$

Anything that follows **identifier** is optional, so once again like for the first choice conflict, we will replace the **LOOKAHEAD** and use square brackets instead:

```
void Nemp_arg_list() : { }
{
    < IDENTIFIER > [< COMMA > Nemp_arg_list()]
}
```

3.2 Removal of Left Recursion:

In this section I will discuss how I removed left recursion from production rules by using **left factoring**.

```
void Expression() : { }
{
    Fragment() Binary_arith_op() Fragment()
|   < LP > Expression() < RP >
|   < IDENTIFIER > < LP > Arg_list() < RP >
|   Fragment()
}
```

The following error was generated, **"Expression... --> Fragment... --> Expression..."**.

When you look at the **Expression()** production rule:

$$\begin{aligned}\langle\text{expression}\rangle \models\ & \langle\text{fragment}\rangle\ \langle\text{binary\_arith\_op}\rangle\ \langle\text{fragment}\rangle\ \mid \\ & (\ \langle\text{expression}\rangle\ )\ \mid \\ & \text{identifier}\ (\langle\text{arg\_list}\rangle\ )\ \mid\end{aligned} \tag{13}$$

And then look at the **Fragment()** production rule:

$$\begin{aligned}\langle\text{fragment}\rangle \models\ & \text{identifier} \mid \textbf{-}\ \text{identifier} \mid \text{number} \mid \textbf{true} \mid \textbf{false} \mid \\ & \langle\text{expression}\rangle\end{aligned} \tag{15}$$

A **Fragment** can be used as a substitute for an **Expression**, so I removed **Expression()** from **Fragment** and using **left factoring** I created a new production rule known as **Basic_expression()** which contains an optional **[Binary_arith_op() Expression()]**:

```
void Expression(): {}
{
    Fragment() Basic_expression()
|   < LP > Expression() < RP > Basic_expression()
}

void Basic_expression() : {}
{
    [Binary_arith_op() Expression()]
}

void Binary_arith_op() : { }
{
    < PLUS >
|   < MINUS >
}

void Fragment() : { }
{
    < IDENTIFIER >[< LP > Arg_list() < RP >]
|   < MINUS > < IDENTIFIER >
|   < NUMBER >
|   < TRUE >
|   < FALSE >
}
```

The next production rule that generated a left recursion error was the **Condition()** production rule:

```
void Condition() : {}
{
    < NOT_OP > Condition()
|   < LP > Condition() < RP >
|   Expression() Comp_op() Expression()
|   Condition() (< OR > | < AND >) Condition()
}
```

This generates a left recursion for **Expression()** and **Condition()**.

To resolve this left recursion, I then had to use left factoring once again and create a new production rule called **Basic_condition()**.

```
void Condition() : {}
{
    (<NOT_OP> Condition() Basic_condition())
|   LOOKAHEAD(3) < LP > Condition() < RP > Basic_condition()
|   (Expression() Comp_op() Expression() Basic_condition())
}

void Basic_condition() : {}
{
   (< OR > | < AND >) Condition() Basic_condition() | {}
}
```

After removing left factoring and choice conflicts from the program, I continued with the remaining production rules.

When finished with the remaining production rules, I ran my code with the following command:

javacc Cal.jj

The program compiled successfully.

4. Testing:

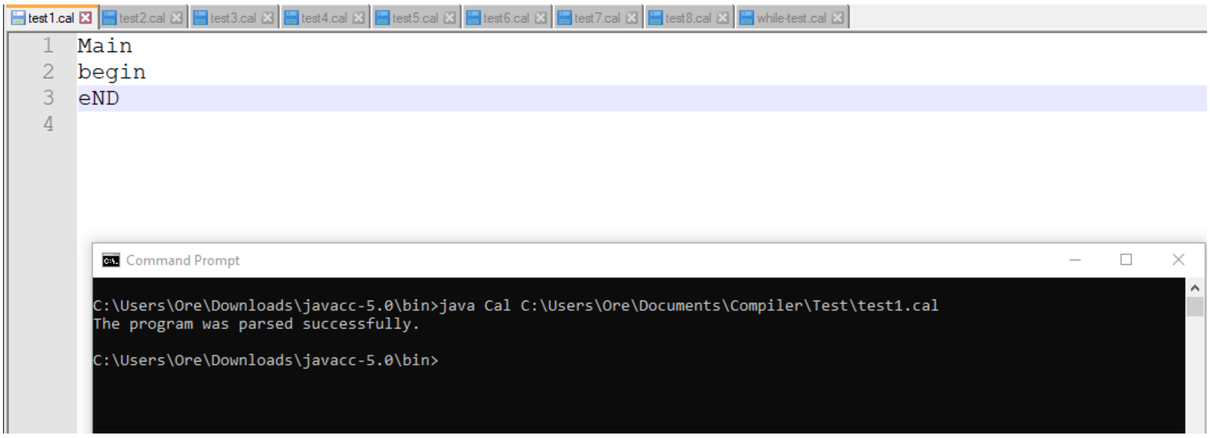
Before I carried out testing, I then ran the following commands:

javac *.java

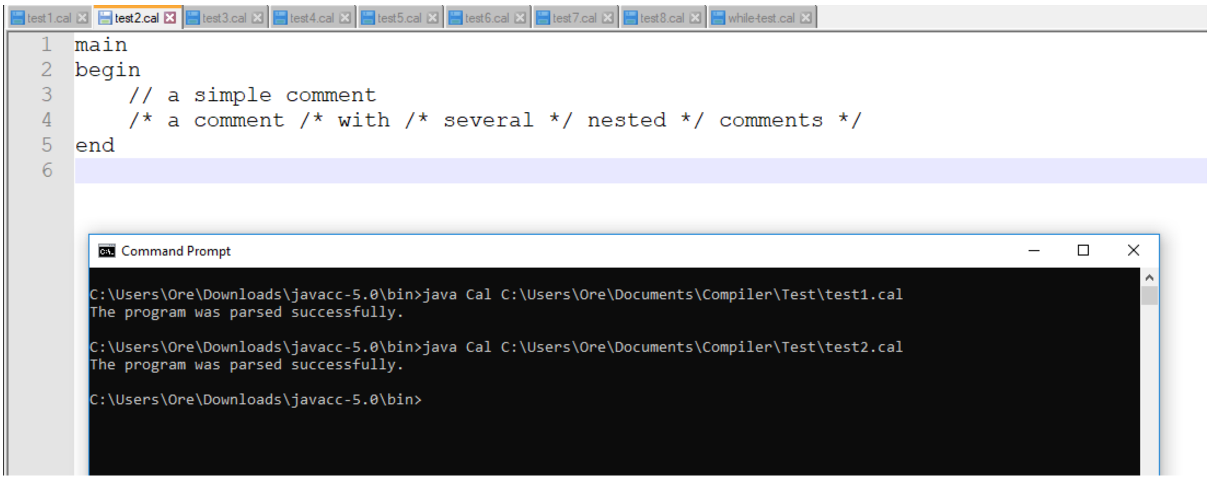
java Cal YOUR_FILE_NAME.cal

I ran my JavaCC code against the example code that was in the Cal documentation.

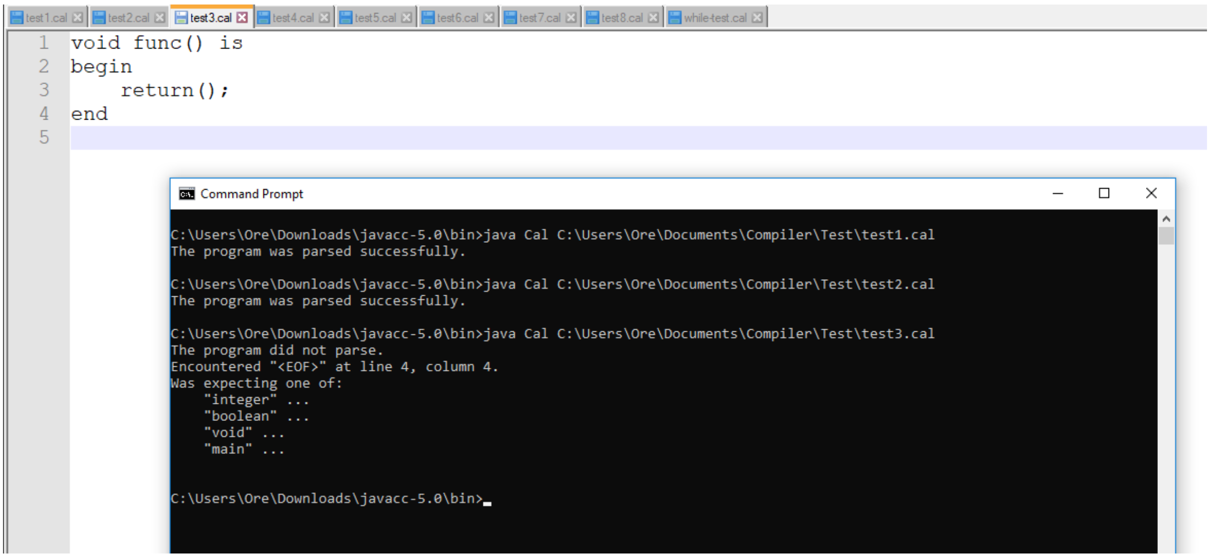
- Test 1 should parser because our program is not case sensitive.



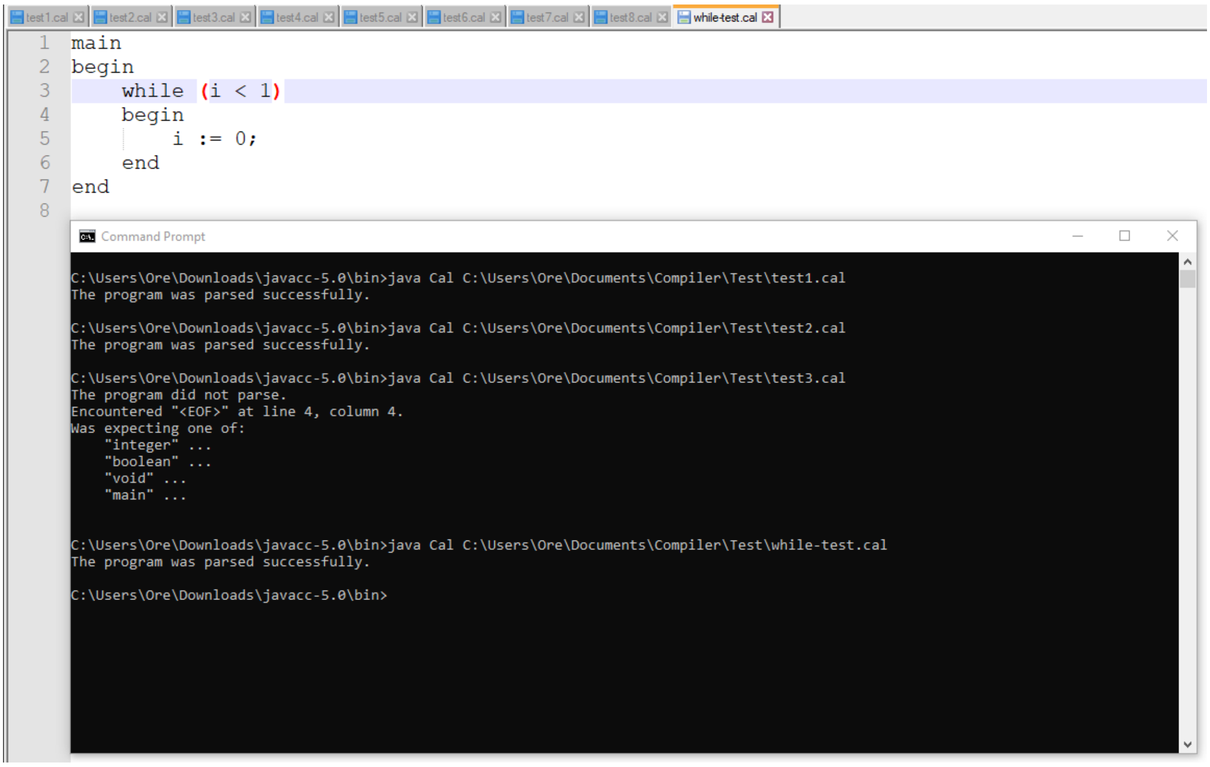
- Test 2 should parser because our program ignores comments.



- Test 3 should fail because our code does not contain a **main**, **begin**, and **end**.



- The following test was simply to demonstrate while loops in the CAL language:



References:

https://www.computing.dcu.ie/~davids/courses/CA4003/CA4003_JavaCC_2p.pdf

https://www.computing.dcu.ie/~davids/courses/CA4003/CA4003_Top_Down_Parsing_2p.pdf

https://www.computing.dcu.ie/~hamilton/teaching/CA423/notes/javacc4.pdf

https://www.computing.dcu.ie/~hamilton/teaching/CA448/notes/JavaCClex4.pdf

https://stackoverflow.com/questions/47122927/how-to-eliminate-this-left-recursion