DCU School of Computing
Assignment Submission

Student Name:	Kyrylo Khaletskyy
Student Number:	15363521
Programme:	BSc. in Computer Applications
Project Title:	Assignment 1: A Lexical & Systax Analyser
Module code:	CA4003
Lecturer:	David Sinclair
Due Date:	10/11/18

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.

Signed: Kyrylo Khaletskyy

Date: 10/11/2018

Introduction

In this report, I will outline the main features of the first assignment for CA4003 Compiler Construction and explain how I obtained my final grammar found in "CALParser.jj". This assignment aims to implement a lexical and syntax analyser using JavaCC for a language called CAL described in the assignment specification.

To begin, I created a folder with the tests given in CAL.pdf. Each test was given a different file, I split them up in order to see which tests passed and which tests failed during my testing phase.

Options & User Code

For this section, I used the code provided on the CA4003 course webpage in JavaCC notes. After a few small changes, it was exactly what I needed in order to test my .cal files. In section 1 of my parser, I outlined that CAL is not a case-sensitive language. In section 2, the parser searches through files found in standard input and executes the grammar using `parser.program()`, afterwards, depending if the parser encounters a "ParseException" it will print either:

`"CAL Parser: CAL program parsed successfully."` or

`"CAL Parser: Encountered errors during the parse."`

Token Definitions

My Token definitions are also similar to the definitions found in the JavaCC notes. For this section, I carefully read the syntax notes provided in the language description and applied them to my language accordingly. Firstly, I entered the keywords and tokens reserved by CAL. Interestingly when I tried to compile my program for the first time I had an error about my "SKP" statement, which at the time was called "SKIP", in JavaCC "SKIP" is a reserved word and cannot be used in my grammar, therefore I renamed it accordingly. Below is a list of keywords and tokens for my parser.

```
TOKEN : /* KEYWORDS */
{
    < VAR : "variable" >
  | < CONST : "constant" >
  | < RETURN : "return" >
  | < INT : "integer" >
  | < BOOL : "boolean" >
  | < VOID : "void" >
  | < MAIN : "main" >
  | < IF : "if" >
  | < ELSE : "else" >
  | < TRUE : "true" >
  | < FALSE : "false" >
  | < WHILE : "while" >
  | < BEGIN : "begin" >
  | < END : "end" >
  | < IS : "is" >
  | < SKP : "skip" >
}
```

```
TOKEN : /* PUNCTUATION */
{
    < COMMA : "," >
  | < SEMIC : ";" >
  | < COLON : ":" >
  | < ASSIGN : ":=" >
  | < LPAREN : "(" >
  | < RPAREN : ")" >
  | < PLUS : "+" >
  | < MINUS : "-" >
  | < NEGATE : "~" >
  | < OR : "|" >
  | < AND : "&" >
  | < EQ : "=" >
  | < NOT_EQ : "!=" >
  | < LT : "<" >
  | < LT_EQ : "<=" >
  | < GT : ">" >
  | < GT_EQ : ">=" >
}
```

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

Next, I implemented Letters, Numbers, Digits and IDs. These were similar to the definitions given in the JavaCC notes but CAL doesn't allow leading 0s unless it was on its own, also the definition for IDs are slightly different. Using regex I allow a number to be either 0, or start with/without a minus sign and specifically say it must start with a number 1-9 (this stops numbers starting with a 0) then followed by any amount of digits denoted by a "*" (the "*" symbol denotes 0+ amount of digits). A simplified version of that grammar is:

NUMBER = 0 | ((-)? [1-9] (0-9)*)

An ID must begin with a letter, afterwards can be followed by any number of letters, digits and underscores. A simplified version of that grammar is:

ID = A(A | B | C)*

Comment skipping was largely handled the same way as the JavaCC notes. The only definition which was missing was a single line comment denoted by a "//". In this scenario, the comment must start with a "//" then continue with any character except "\n" and "\r", and must end with a newline character. Initially, I only had "\n" but after a closer look through lexical analysis notes, I found that "\r" must also be implemented as it is a different type of newline character.

Single-Line Comment = "//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")

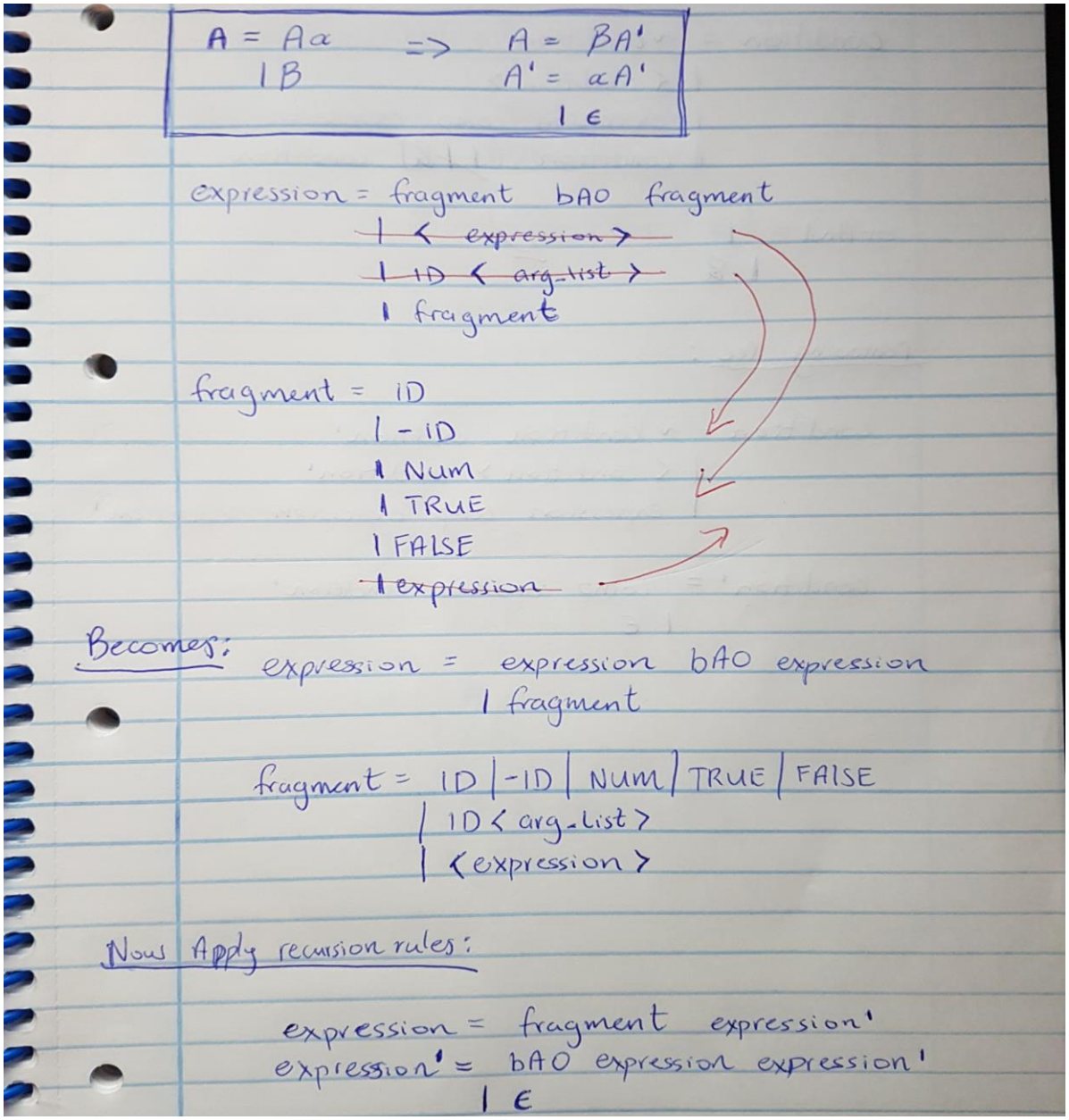
Eliminating Left Recursion

When implementing the grammar the first step I took was carefully writing out the definitions found in the CAL.pdf. After writing them out and fixing any small errors, I was faced with the two main "Left recursion detected" problems, these were in "expression" and "condition".

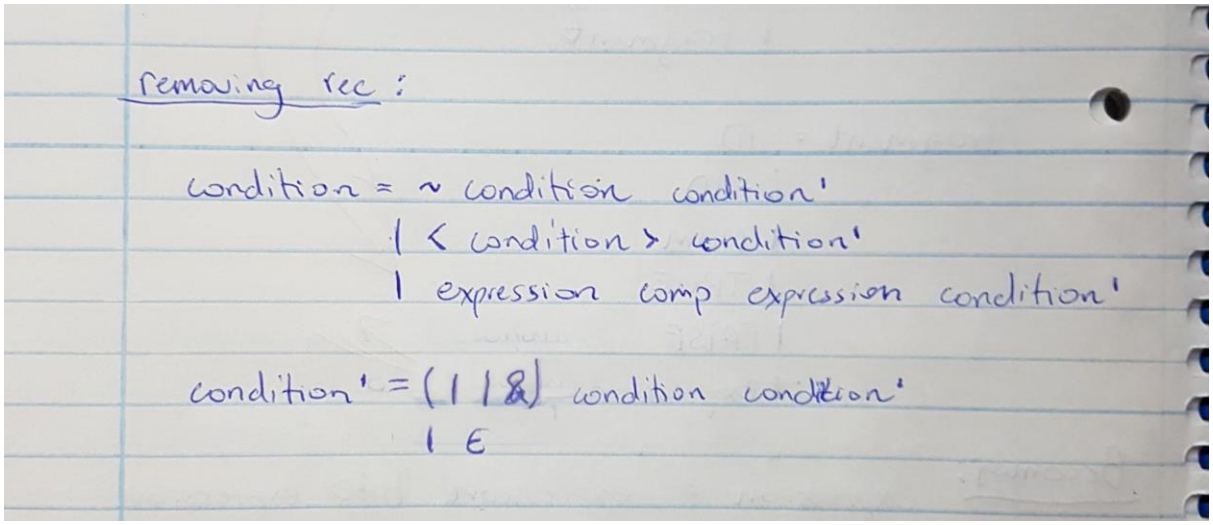
```
Java Compiler Compiler Version 5.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file CALParser_v1.jj . . .
Error: Line 227, Column 1: Left recursion detected: "expression... --> fragment... --> expression..."
Error: Line 248, Column 1: Left recursion detected: "condition... --> condition..."
Detected 2 errors and 0 warnings.
```

When solving recursion problems, I found that working on paper was very beneficial as I could rearrange, and solve problems in a simplified manner. When eliminating left recursion I found the notes on top-down parsing useful, I applied a similar method when eliminating recursion in my grammar. I have also found some extra useful information on the following site:

http://www.engr.mun.ca/~theo/JavaCC-FAQ/javacc-faq-moz.htm



I started with expression, before removing left recursion I needed to rewrite it, since fragment contains expression(), I can turn fragment() binary_arith_op() fragment() to expression() binary_arith_op() expression(), and the singular fragment() will remain in expression in order to call the other tokens. <LPAREN> expression() <RPAREN> and <ID> <LPAREN> arg_list() <RPAREN> can be moved to fragment, this is because I can call fragment from expression. Now that both are rearranged, it makes it easier to remove left recursion. I apply the rule given in the notes (can be seen at the top of the picture above). I take out the expression() binary_arith_op() expression() and move it into a new expressionPrime method removing the first token and adding expressionPrime() | {} at the end. Then expressionPrime is added at the end of each line in expression. A new version of expression and expressionPrime is shown on the bottom of the picture above. This leaves me with a choice conflict within fragment, but I will come back to this after I solve the next recursion problem.

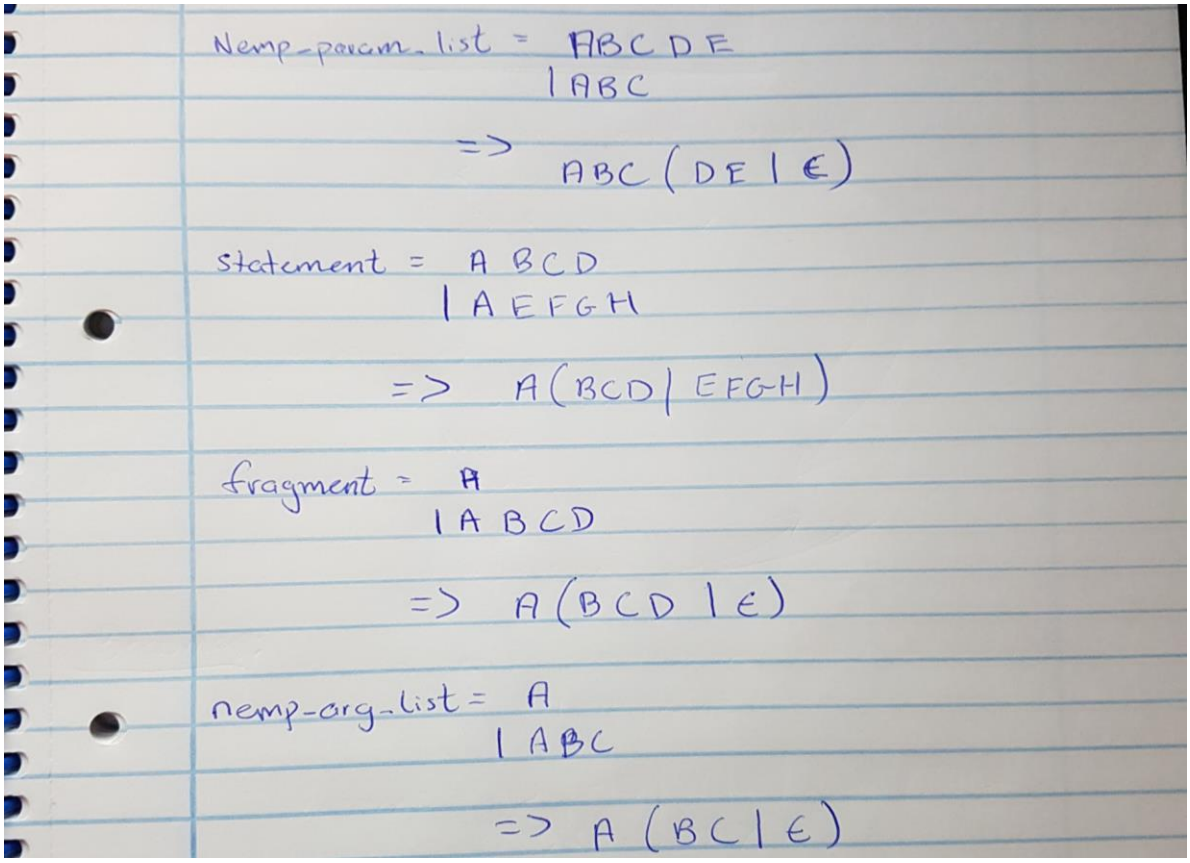


I applied the same principles to condition as I did when removing left recursion in expression. I made a new method called conditionPrime, and moved the condition() (<OR> | <AND>) condition() into the new method, removing the first token, adding conditionPrime() | {} to the end. Then adding conditionPrime() at the end of each line in the condition method.

Eliminating Choice Conflicts

```
Reading from file CALParser_v1.jj . . .
Warning: Choice conflict involving two expansions at
         line 200, column 5 and line 201, column 5 respectively.
         A common prefix is: <ID> ":"
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 218, column 5 and line 219, column 5 respectively.
         A common prefix is: <ID>
System Settings der using a lookahead of 2 for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 247, column 5 and line 248, column 5 respectively.
         A common prefix is: "(" <ID>
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 252, column 6 and line 253, column 5 respectively.
         A common prefix is: "(" <ID>
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 282, column 5 and line 283, column 5 respectively.
         A common prefix is: <ID>
         Consider using a lookahead of 2 for earlier expansion.
File "TokenMgrError.java" is being rebuilt.
```

Before applying the new methods to my grammar I made a backup file, this would prevent me from having to write it out again in case something went wrong. After entering the new grammar into my CALParser.jj file I found that there were many choice conflicts, as shown in the screenshot above.



To eliminate choice conflicts I simplified the corresponding methods on paper and took out the common token. These were mainly split up into two types of eliminations:

1. Removing the common token and enclosing the other tokens in brackets in order to give a choice of either one or the other line. For example:
   X = A B C | A D E

   Becomes: X = A (B C | D E)

2. When you have a singular token conflicting with the first token of another line, I removed the common token, enclosed the other line with it | {}. For example:
   X = A B C | A

   Becomes: X = A (B C | {})

The last issue I faced was a choice conflict between <LPAREN> condition() <RPAREN> conditionPrime() in condition and <LPAREN> expression() <RPAREN> in fragment. I found this section most challenging.

```
void condition() : {} {
    <NEGATE> condition() conditionPrime()
  | <LPAREN> condition() <RPAREN> conditionPrime()
  | expression() (comp_op() expression()
                | <LPAREN> expression() <RPAREN>) conditionPrime()
}
```

In my initial attempt shown above, I moved <LPAREN> expression() <RPAREN> into condition, and then grouped it with expression() comp_op() expression() in order to solve further choice conflict with <ID> in fragment. While the tests return as successful and the parser doesn't show any compile errors, I have later found that expression can be run without condition, therefore moving it into condition would be illegal.

Due to this, I was forced to use a lookahead. Initially, I tried fixing the choice conflict using LOOKAHEAD(2), but upon further testing, I have found that no fixed amount of lookahead will work properly for all test cases. If a condition that starts with "(((((((1" you can't tell if that parenthesis is surrounding a condition or an expression without looking further ahead. Therefore I used a syntactic lookahead. Implementing LOOKAHEAD(expression() comp_op()) expression() comp_op() expression() conditionPrime() removed all compile errors, and now my parser passes all my tests successfully.

Conclusion & Testing

```
zarrexx@ubuntu:~/assignment1$ java CALParser < test/insensitive1.cal
Reading from standard input...
CAL Parser: CAL program parsed successfully.
zarrexx@ubuntu:~/assignment1$ java CALParser < test/insensitive2.cal
Reading from standard input...
CAL Parser: CAL program parsed successfully.
zarrexx@ubuntu:~/assignment1$ java CALParser < test/insensitive3.cal
Reading from standard input...
CAL Parser: CAL program parsed successfully.
zarrexx@ubuntu:~/assignment1$ java CALParser < test/comments.cal
Reading from standard input...
CAL Parser: CAL program parsed successfully.
zarrexx@ubuntu:~/assignment1$ java CALParser < test/scopes.cal
Reading from standard input...
CAL Parser: CAL program parsed successfully.
zarrexx@ubuntu:~/assignment1$ java CALParser < test/functions1.cal
Reading from standard input...
CAL Parser: CAL program parsed successfully.
zarrexx@ubuntu:~/assignment1$ java CALParser < test/functions2.cal
Reading from standard input...
CAL Parser: CAL program parsed successfully.
```

Upon completion of this project, I have learned a lot about JavaCC and the elimination of left recursion and choice conflicts. While I was unable to fix the last choice conflict, all the attempts I made trying to eliminate it taught me a lot about compilers. After finishing all the changes to my grammar I performed some final testing, all tests I ran were successful as shown in the screenshot above.

In order to run my parser the following steps are taken:

javacc CALParser.j
javac *.javaj
java CALParser < "test file"