Declaration on Plagiarism

Assignment Submission Form

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

Name(s): Declan Lunney
Programme: Computer Applications
Module Code: CA4003
Assignment Title: Assignment 1: A Lexical and Syntax Analyser
Submission Date: 10/11/18
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): Declan Lunney **Date:** 10/11/18

Table of Contents

Introduction

The aim of this assignment is to implement a lexical and syntax analyser using JavaCC for a simple language called CAL. In this report I will outline I implemented the major components for creating a lexical and syntax analyser for the CAL language.

I started off the assignment using the help of javacc notes from our lectures at:
http://www.computing.dcu.ie/~davids/courses/CA4003/CA4003_JavaCC_2p.pdf

1 Options

The parser will have options. I made IGBORE_CASE = TRUE as the language is not case sensitive. Although I did try keep my variable names following same format as it's good practice in coding that I learned in comparative languages in third year. Also set JAVA_UNICODE_ESCAPE to true as it can interpret the Unicode when it normally doesn't.

```
/**********************************
***** SECTION 1 - OPTIONS *****
**********************************/

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

2 User code

I used what was in the notes to start my parser begin and end code (**user code**). After this I defined my tokens. All javacc parsers must begin with a declaration with its parser name, in this case it is.
PARSER_BEGIN(CALParser)
My 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 and to make sure it adhered to the CAL language.

3 Tokens

3.1 Comment handling

Firstly, I began by declaring the comment tokens which are to be the ignored:

```
SKIP:
{
" "| "\n"| "\r"| "\t"| "\f"
}

TOKEN_MGR_DECLS :
{
  static int commentNesting = 0;
}

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

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

3.2 Keywords, Operators & Relations

Declaring keywords, operators and relations was self-explanatory to do this. It was simply token blocks with the name of the token followed by the string that matches to it.

```
TOKEN: {
  <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">
| <SKIPY: "skip">
}
```

```
TOKEN: {
 <COMMA: ",">
| <SEMICOLON: ";">
| <COLON: ":">
| <ASSIGNED_VAL: ":=">
| <LEFT_BRACKET: "(">
| <RIGHT_BRACKET: ")">
| <PLUS: "+">
| <MINUS: "-">
| <TYLDA: "~">
| <OR: "|">
| <AND: "&">
| <EQUAL: "=">
| <NOT_EQUAL: "!=">
| <LESS: "<">
| <LESS_OR_EQUAL: "<=">
| <GREATER: ">">
| <GREATER_OR_EQUAL: ">=">
```

3.3 Identifiers & Numbers

I inserted the reserved words in as tokens as well as the keywords and punctuation. When creating the IDENTIFIER (etc. Variable name) the rule was that it began with a letter followed by either a number or a DIGIT (0 – 9) where the grammar would be set to <LETTER> (<LETTER> | <DIGIT>)*>. All real numbers can be either negative or plus so I had to put the (–) as optional. I set it choose an

empty set so it could left the - as optional <INT:(<MINUS>)? (<DIGIT>) (<DIGIT>)*>

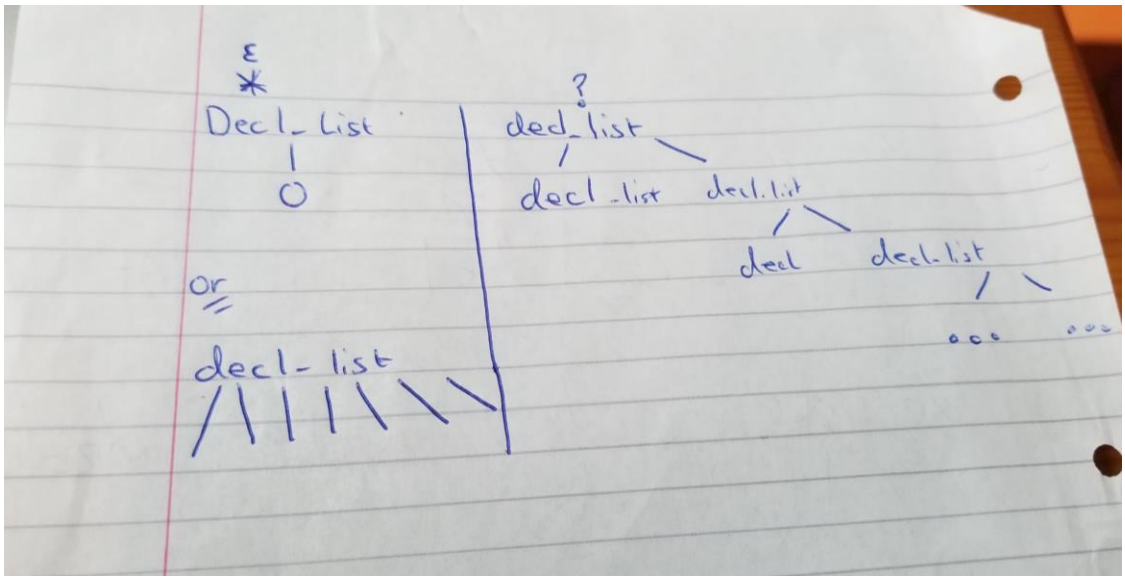
```
| <INT:(<MINUS>)? (<DIGIT>)+>
| <ID:<LETTER> (<LETTER> | <DIGIT>)*>
| <#LETTER: ["A"-"Z", "_", "a"-"z"]>
| <#DIGIT: ["0"-"9"]>
}
```

4 Grammar

One of the first changes I had to make was to the Decl function which had a Kleene star surrounding it. This Kleene star meant that in other functions where it was called with a Kleene star an error would occur as a result of it being able to parse to an empty string. I replaced the star with **?. (A)* = zero or more A vs (A)? Zero or one A.**

At any choice point, the parser chooses the first path that has a valid match.

If the parser is trying a path and that path is valid for the empty set ()?, then it is always chosen.



4.1 Left recursion

For the next part in defining my grammar, I will outlay the difficulties I ran into and how I overcame them. When working on the syntax several conflict choices and left-recursion errors arose. So when I completed the grammar the first thing I noticed was left recursion from not only the condition and expression terminals. From lectures I knew that Top-Down parsers can't handle left recursion in the grammar. So I added new non terminal condition_remove_lr(). I used the lectures notes on how to remove left recursion by:

Eliminating Left-recursion

To remove left-recursion, we can transform the grammar.

Consider the grammar fragment:

$$\begin{array}{lcl} A & ::= & A\alpha \\ & | & \beta \end{array}$$

where α and β do not start with A.
We can rewrite this as:

$$\begin{array}{lcl} A & ::= & \beta A' \\ A' & ::= & \alpha A' \\ & | & \epsilon \end{array}$$

where A' is a new non-terminal

This fragment contains no left-recursion.

```
void condition() :
{}
{
   <TYLDA> condition() condition_remove_lr()
| <LEFT_BRACKET> condition() <RIGHT_BRACKET> condition_remove_lr()
```

```
// New non-terminal created when removing left recursion (prime version of same thing)
void condition_remove_lr() :{}
{
   ((<OR> | <AND>) condition() condition_remove_lr())|
   {return;} //check

}
```

The red dotted line is showing what was transformed of the non terminal after I removed left recursion. This eliminated the Left recursion from the condition non – terminal. To remove the left recursion in expression() I arranged the non- terminals in order and substituting for expression() in

fragment()s sentential form. When I substituted expression in for fragment non-terminal, fragment was transformed. After I did the substitution I was faced with direct left recursion involving the two fragments().

To eliminate this I simply removed the left recursion in the same way I used to remove the left recursion from the condition() non-terminal where I created a new non- terminal.

After I eliminated the indirect left recursion I could run the grammar. To reduce lookaheads this was added to allow the expression to compare its productions and chose its paths before calling the fragment non-terminal.

```
void expression() :{}
{
   <ID>((<LEFT_BRACKET> arg_list() <RIGHT_BRACKET> | fragment_prime())(<LEFT_BRACKET> arg_list() <RIGHT_BRACKET> fragment_prime())?)
  | (<MINUS> <ID> fragment_prime()| <INT> fragment_prime()| <True> fragment_prime()| <False> fragment_prime() | <LEFT_BRACKET> expression() <RIGHT_BRACKET> fragment_
}
```

```
void fragment() :
{}
{
  <ID>( <LEFT_BRACKET> arg_list() <RIGHT_BRACKET> fragment_prime()|fragment_prime())| <MINUS> <ID> fragment_prime()| <INT> fragment_prime()| <True> fragment_prime()|
}


// new non terminal was created when getting rid of left recursion.
void fragment_prime() :
{}
{
 (biary_arith_op() fragment())?
}
```

4.2 Lookaheads and warnings

There were multiple choice conflicts, two of which were solved by transforming nemp_parameter_list() and nemp_arg_list().

```
244  void piece() :
245  {}
246  {
247    <ID>( <LEFT_BRACKET> arg_list() <RIGHT_BRACKET> piece_prime()|piece_prime())| <MINUS> <ID> piece_prime()| <INT> piece_prime()| <True> piece_prime()| <False> piece
248  }
```

To get rid of lookaheads, I separated my two functions and looked at one problem at a time. I worked with the errors I was retrieving in the expression() non terminal first. Where I was retrieving trouble in the expression() non terminal it was mainly to do with the fragment() non-terminal.

I rearranged the productions and placed them in expression() and this stopped the productions reaching down to fragment. As a result warnings were not recieved. After studying the code, I could eliminate some of the grammar as it was being already used in the new non-terminal`s I created to eliminate the left recursion.
The grammar in yellow outline box above was inserted into the expression non-terminal, to eliminate the warnings thus reducing the lookaheads. This allowed expression to compare its productions and chose its paths before calling the fragment non-terminal.
In the condition nonterminal, I used the same principle where I substituted some of the grammar in fragment into condition to avoid the conflicts.

4.3 Grammar elimination

I was able to remove some of the grammar as it was already being called in other non-terminals. This also helped to get rid of warnings and reduce the lookaheads.
I substituted fragment into expression to remove lookaheads and warnings. I was able to eliminate the grammar because I had substituted the grammar of fragment into the expression already. When removing the left recursion I had created a non-terminal fragment_prime() that contained binary_arith_op() and called fragment() which was called recursively. This allowed me to eliminate this piece of grammar.

```
void fragment() :
{}
{
  <ID>( <LEFT_BRACKET> arg_list() <RIGHT_BRACKET> fragment_prime()|fragment_prime())
}


// new non terminal was created when getting rid of left recrusion.
void fragment_prime() :
{}
{
 (biary_arith_op() fragment())?
}
```

5 Debugging and testing

It is useful to know what choices JavaCC has made when parsing the input. This helps in debugging the grammar, when things do not work as you expect. What I did was debug from the inside out so once my grammar was completed and it compiled successfully, I started at the very low level of comp_op. So, I made sure all the operators worked first then I worked up. This proved good as I did not have call parser program, I could just call the specific function and when I ran into errors I know exactly where it is that causing the issue.
Although my LL(1) grammar was passing all four tests that were given in the CAL document, I wanted to make sure that it would throw errors if there was an error. I included 3 error tests below.

Put in 0.00 instead of 0

```
Declans-MacBook-Pro:compiler declanlunney$ java CALParser test.txt
Exception in thread "main" ParseException: Encountered " <OTHER> ". "" at lin
18, column 34.
Was expecting one of:
    "begin" ...
    "+" ...
    "-" ...
    "|" ...
    "&" ...
```

Put in &&

```
Declans-MacBook-Pro:compiler declanlunney$ java CALParser test.txt
Exception in thread "main" ParseException: Encountered " "&" "& "" at line 125,
column 39.
Was expecting one of:
    "true" ...
    "false" ...
    "(" ...
    "-" ...
    "~" ...
    <INT> ...
    <ID> ...
```

Taking away symbol + to see what it the program is expecting, and it correlates to the CAL document correctly.

```
Declans-MacBook-Pro:compiler declanlunney$ java CALParser test.txt
Exception in thread "main" ParseException: Encountered " <INT> "0 "" at line 118
, column 31.
Was expecting one of:
    "(" ...
    "+" ...
    "-" ...
    "=" ...
    "!=" ...
    "<" ...
    "<=" ...
    ">" ...
    ">=" ...
```

5.1 Running the parser

To run the Syntax analyser you simply call the interpreter as follows: `javacc CALParser.jj`

```
Declans-MacBook-Pro:compiler declanlunney$ javacc CALParser.jj
Java Compiler Compiler Version 5.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file CALParser.jj . . .
File "TokenMgrError.java" is being rebuilt.
File "ParseException.java" is being rebuilt.
File "Token.java" is being rebuilt.
File "JavaCharStream.java" is being rebuilt.
Parser generated successfully.
Declans-MacBook-Pro:compiler declanlunney$
```

Then `javac *.java`

Finally, to test I typed `java CALParser (TESTFILE)`

```
Declans-MacBook-Pro:compiler declanlunney$ java CALParser test.txt
Completed Successfully
Declans-MacBook-Pro:compiler declanlunney$
```