Assignment Submission

Module Title	Compiler Construction
Module Code	CA4003
Assignment Title	A Lexical and Syntax Analyser for the CCAL Language
Submission Date	7th November 2016
Name	Tríona Barrow
Student Number	11319851
Module Co-Ordinator	David Sinclair

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 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 have read and understood the Assignment Regulations. I 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 or any other person for assessment on this or any other course of study.

A Lexical and Syntax Analyser for the CCAL Language

I started with the basic program structure from the notes in JavaCC on the module homepage, to begin with. This gave me the basic bones to build off specifically for CCAL. I'll break down the changes made section by section.

- *Options*

As CCAL is not case specific, I added in a flag here for IGNORE_CASE and set it to true - this will ensure that there are no issues in tokenising.

- *User Code*

I renamed this section to TokenParser, and changed any references to that within here. This is really the only change I made in this section - as the version from the notes did not need to be modified.

- *Token Definitions*

Here is where a lot of the different reserved words and operators had to be defined in order to analyse and parse CCAL correctly.
The main issue I ran into with this section was trying to define ε. Initially, I defined it as a token as `""` - however this ended up causing choice conflicts within the compiler. I ended up searching online, and found this segment of code resolved that error (from this StackOverflow thread).

```
TOKEN : {
    < EMPTY : "" > : MATCH_NON_EMPTY
}

<MATCH_NON_EMPTY> TOKEN : {
    < NON_EMPTY : ~[] >
}
```

Keywords and punctuation were also pretty straightforward, I just had to ensure that the "skip" token was named something else - as `SKIP` was already a reserved keyword for JavaCC. I ended up just naming is `TKNSKIP` for convenience.
We did not have to worry about strings or floats, only integers and negative integers. Integers required some special handling, because they could not start with a 0 - so I created a expression within the number token definition for the start digit. This section ended up as follows:

```
TOKEN : /* Numbers */ {
      < NUM : (<UNARY_MINUS>)? (<STARTDIGIT>)+ (<DIGIT>)* >
    | < #STARTDIGIT : ["1" - "9"] >
    | < #DIGIT : ["0" - "9"] >
}
```

- *The Grammar*

This was really the most difficult section to puzzle out.

Here was where certain definitions between methods caused leftmost recursion between `condition()` and `expression()`, and again between `expression()`, `fragment()` and back to `expression()` again.

The first occurrence of leftmost recursion was the quickest to resolve out of the two - I ended up moving most of the definitions for `condition()` into a new method `base_condition()`, and then called `base_condition()` from `condition()`. (Essentially breaking it out into two methods, the base case and the recursive case.) The code looked as follows:

```
void condition() : {} {
    base_condition() (<LOGOR> | <LOGAND>) base_condition()
}

void base_condition() : {} {
      <TILDE> condition()
    | <LBR> condition() <RBR>
    | expression() comp_op() expression()
}
```

The second occurrence took more puzzling out - however ended up being a simple fix. From searching online again, the easiest way to break out of this recursion was to embed the expression() statement within fragment() in bracket tokens - as if it got passed back to this point, it would within the program be within brackets. (From this StackOverflow thread.) The code ended up like below (after some cleanup for choice conflicts too):

```
void expression() : {} {
      fragment() (binary_arith_op() fragment())?
}

void fragment() : {} {
      (<MINUS_SIGN>)? <ID>
    | <NUM>
    | <TRUE>
    | <FALSE>
    | <LBR> expression() <RBR>
}
```

Apart from this - there was one choice conflict that I was unable to resolve without using `LOOKAHEAD`. From the definitions of `base_condition()` and `fragment()` - in both lines `<LBR>` (`"("`) was the first token and was causing a choice conflict. I ended up resolving this by putting in `LOOKAHEAD` within `base_condition()` like so:
`LOOKAHEAD(2) <LBR> condition() <RBR>`

Below is the code without the `LOOKAHEAD` included, with the offending lines highlighted. I could not find an alternative way around this myself, and ran out of time trying.

```
void expression() : {} {
      fragment() (binary_arith_op() fragment())?
}

void fragment() : {} {
      (<MINUS_SIGN>)? <ID>
    | <NUM>
    | <TRUE>
    | <FALSE>
    | <LBR> expression() <RBR>
}
```

```
void condition() : {} {
    base_condition() (<LOGOR> | <LOGAND>) base_condition()
}

void base_condition() : {} {
      <TILDE> condition()
    | <LBR> condition() <RBR>
    | expression() comp_op() expression()
}
```

Otherwise, the code does seem to work, and generates an output in command line from the sample code in the specification:

```
integer Identifier(multiply) ( Identifier(x) : integer ,
Identifier(y) : integer ) { var Identifier(result) : integer ;
var Identifier(minus_sign) : boolean ; if ( Identifier(x) < 0 &&
Identifier(y) >= 0 ) { Identifier(minus_sign) = true ;
Identifier(x) = - Identifier(x) ; } else { if Identifier(y) < 0
&& Identifier(x) >= 0 { Identifier(minus_sign) = true ;
Identifier(y) = - Identifier(y) ; } else { if ( Identifier(x) < 0
) && Identifier(y) < 0 { Identifier(minus_sign) = false ;
Identifier(x) = - Identifier(x) ; Identifier(y) = - Identifier(y)
; } else { Identifier(minus_sign) = false ; } } }
Identifier(result) = 0 ; while ( Identifier(y) > 0 ) {
Identifier(result) = Identifier(result) + Identifier(x) ;
Identifier(y) = Identifier(y) - Number(1) ; } if
Identifier(minus_sign) == true { Identifier(result) = -
Identifier(result) ; } else { skip ; } return (
Identifier(result) ) ; } main { var Identifier(arg_1) : integer ;
var Identifier(arg_2) : integer ; var Identifier(result) :
integer ; const Identifier(five) : integer = Number(5) ;
Identifier(arg_1) = Number(-6) ; Identifier(arg_2) =
Identifier(five) ; Identifier(result) = Identifier(multiply) (
Identifier(arg_1) , Identifier(arg_2) ) ; }
```

- *Links Referenced*

http://stackoverflow.com/questions/11171486/epsilon-definition-in-javacc
http://stackoverflow.com/questions/20482797/left-recursion-detected-in-jj-file