Submission Liam McAweeney 14415152

Programme: Computer Applications Module Code:

CA4003 Assignment Title: Lexical and Syntax Analyser for CCAL Language

Submission Date: 12/11/2018

Module Coordinator: David Sinclair

Name(s): Liam McAweeney Date: 12/11/2018

CA4003 - Compiler Construction Assignment 1

Student Name: Liam McAweeney
Student Number: 14415152
Submission Date: 09/11/18

Content

Lexical Analysis

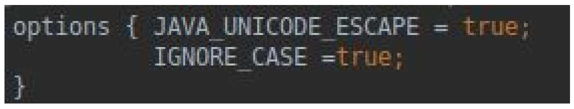
Design

Lexical Analysis is the process of converting a stream of characters into a stream of tokens. A token is a string or character which is recognised by the language, such as keywords, identifiers or parentheses.
Some examples:
    “main”, “void”, “tmp”, “{“, ”)”
Removing white spaces makes parsing a lot easier.
To achieve this I am going to use regular expression.
Regular expression or regex is used to find a sequence of characters known as a pattern.

Implementation

This section corresponds to section 3 - TOKEN DEFINITIONS in myparser.jj

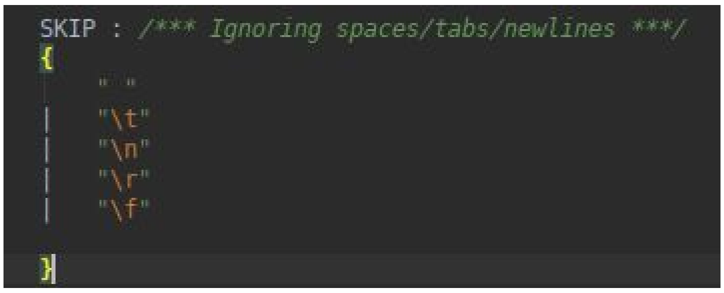
As stated in the language description, the parser has to be case insensitive for keywords such as main. To achieve this I set an option at the top of the file like so:



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

Following this my first task I have is to allow the parser to ignore white spaces and comments.
As mentioned above, I am using regular expression to achieve this. I create a pattern which is recognised by *myparser.jj* as a pattern to skip. The pattern can be any of the following, whitespace, tab, return, new line or a form feed. This is define using alternation which is defined using the “ | ” symbol. Alternation is the similar to the binary operation “OR”.

Below is the definition of one of my skip patterns:

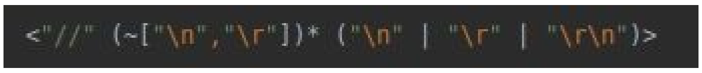


```
SKIP : /*** Ignoring spaces/tabs/newlines ***/
{
    " "
|   "\t"
|   "\n"
|   "\r"
|   "\f"

}
```

The next pattern I defined was to skip comments. As described in the language definition, there is two types of comments:

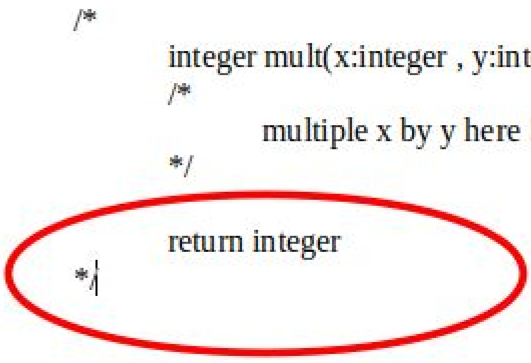
1. A single line comment - This pattern starts with “//” and must end with a newline or return. Between its bonding tokens it can contain any amount of any characters except newline and return:



```
<"//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")>
```

The symbol “~” means not, as you can see it is also wrapped in (...)*. This is regex’s Kleene Closure operation which denotes zero or more occurence of everything inside.

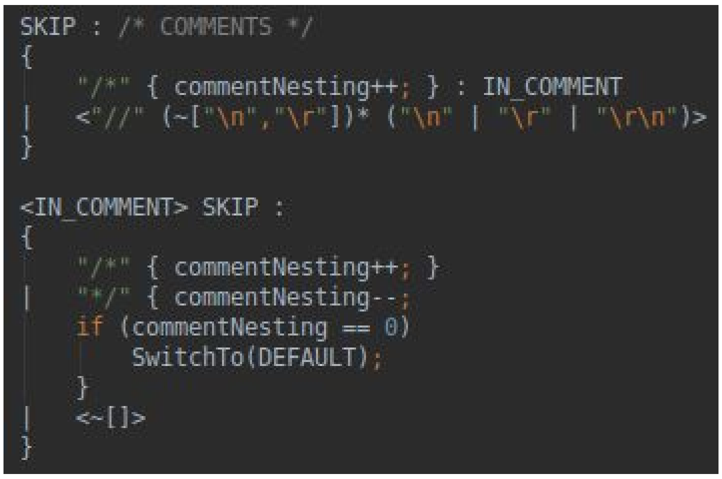
2. Multi-line comment - This pattern starts with “/*” and ends with “*/”. The type of comment is different from above as it allows for nested comments, This could cause problems as may end the comment token to early.



```
/*
        integer mult(x:integer , y:integer) is
        /*
                multiple x by y here !
        */

        return integer
*/
```

“Return integer” would not be recognised to skip. To combat this I count the number of opening tokens and every time I reach a closing token I decrement the count. When the count reaches zero the comment is complete.



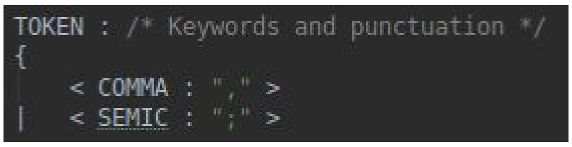
```
SKIP : /* COMMENTS */
{
    "/*" { commentNesting++; } : IN_COMMENT
|   <"//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")>
}

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

Next I defined reserved words from the language's description. Examples of keywords are **variable**, **constant**, **return**. This language has defined **skip** as a reserved word. Javacc has its own set of reserve words one of which is **skip**, so i have to define this token as SKIPSTREAM

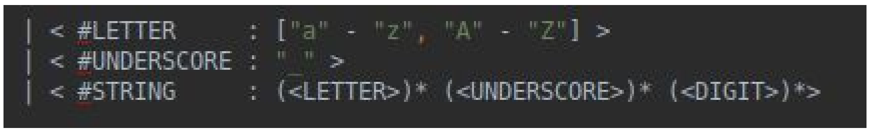
Next I defined the the language's punctuation. Examples of the punctuation are
, ; : :=

Each word or punctuation of the two above are given a unique token. This differs from the comments and whitespaces as the parser doesn't need to recognised them differently, since they are all being discarded. Example of the define tokens:



```
TOKEN : /* Keywords and punctuation */
{
    < COMMA : "," >
|   < SEMIC : ";" >
```

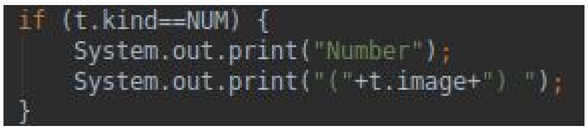
Lastly I define the language's identifiers and numbers. To do this I defined some characteristics of each. I created a token for zero and underscore for example. Neither of these tokens could be recognised outside defined identifier or number, this was achieved by placing "#" in front of the token's ID.



```
| < #LETTER     : ["a" - "z", "A" - "Z"] >
| < #UNDERSCORE : "_" >
| < #STRING     : (<LETTER>)* (<UNDERSCORE>)* (<DIGIT>)*>
```

Test

To test the the lexical analyser I created a print statement for each token.
I created a test file which contain all the punctuations and reserved words.
When a token was read the print statement would print out the expected token id and the stream which was read in.



```
if (t.kind==NUM) {
    System.out.print("Number");
    System.out.print("("+t.image+") ");
}
```

I used a for loop to iterate through the tokens in the text file, using *getNextToken()* until EOF. (End Of File).

Syntax Analysis

Design

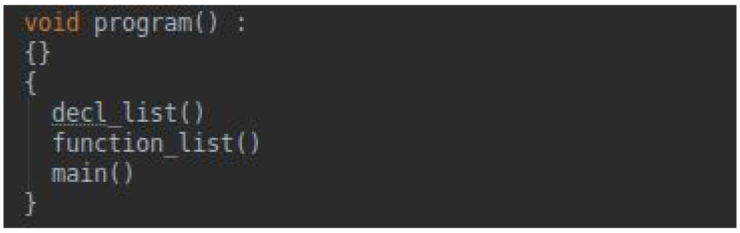
Syntax Analysis is the process of taking a stream of tokens, which have been created by the lexical analyser described above, to form a valid "sentence". This is completed by creating a set of rules which define the order in which.

Implementation

This section corresponds to section 4 - THE GRAMMAR & PRODUCTION RULES in myparser.jj

A valid grammar is defined using production rules. Production rules look very similar to functions. The production rules are made from the tokens, defined in the section above, regular expression and other production rules.

The first production rule defined is the most high level rule:

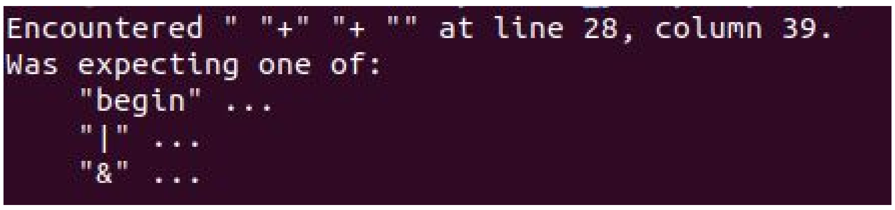


```
void program() :
{}
{
  decl_list()
  function_list()
  main()
}
```

program is made from three other production rules, decl_list, function_list and main. Each of these rules must parse without throwing any errors.

The parsing function begins with the lexical analyser creating a stream of tokens which is then passed to the program production rule, from there the each production rule may be reach. If any throw an error the program

will fail and will describe where the error occurs and say which token it encountered and which it expected.
An example of such is:



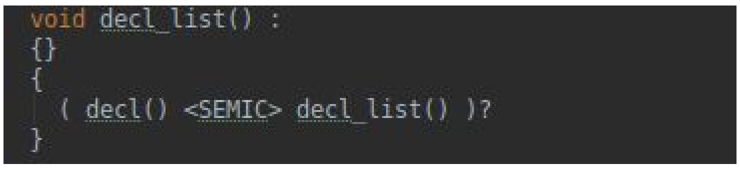
```
Encountered " "+" "+ "" at line 28, column 39.
Was expecting one of:
    "begin" ...
    "|" ...
    "&" ...
```

This is the condition that caused the error: if (x < 0) + (y < 0)
A binary add is not allowed after a condition, only a logical OR, AND or the keyword begin.

In the language description decl_list is defined as:

$$\langle \text{decl\_list} \rangle \models (\langle \text{decl} \rangle \,;\, \langle \text{decl\_list} \rangle \mid \epsilon)$$

ϵ denotes an empty set which in this context represents nothing, allowing the rule to not accept any tokens and not fail. The above rule also contains regex's alternation, either the left side or the right side must be match. This can be simplified using by wrapping the left side in brackets and placing a "?" at the end. This is another regex operation which means zero or one occurences of the enclosed rule can occur.

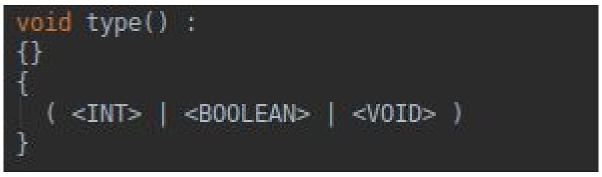


```
void decl_list() :
{}
{
  ( decl() <SEMIC> decl_list() )?
}
```

<SEMI> is the token which denotes ";".

As you can see decl_list is recursive as it can call itself. This is ideal as you this ensures that decl_list has a set number of children. It will always have two children or none. This is very important when creating an Abstract syntax tree in assignment 2.

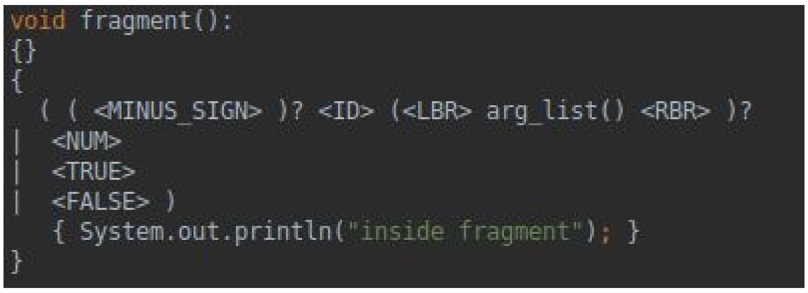
When starting the implementation of the syntax analyser I started with the lowest level product rules, these are the production rules which do not call other production rules. This allowed me to build from the bottom up, which I believe made the implementation process simpler.



```
void type() :
{}
{
  ( <INT> | <BOOLEAN> | <VOID> )
}
```

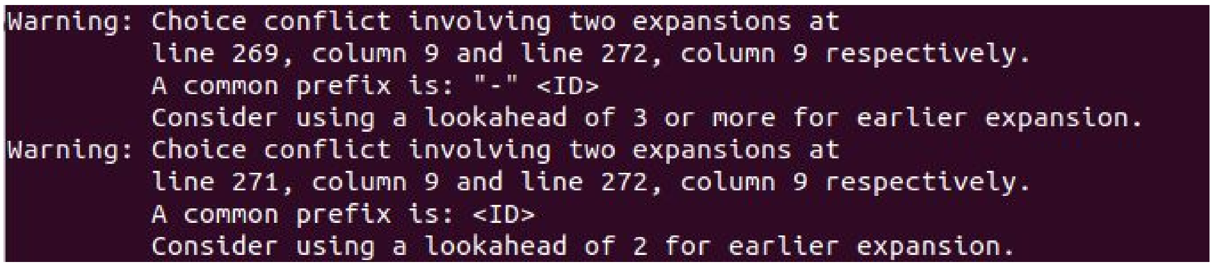
I then worked through all the rules set in the language's description and completed the analyser.

I inserted multiple print statements throughout the production rules to follow the path in which the parser was entering. This helped greatly in the discovery of ambiguities in my rules.



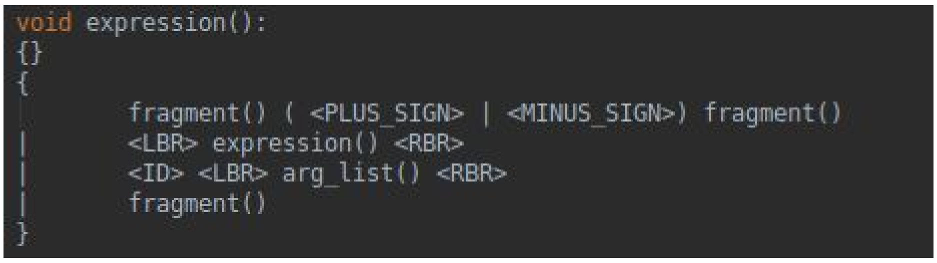
```
void fragment():
{}
{
  ( ( <MINUS_SIGN> )? <ID> (<LBR> arg_list() <RBR> )?
|   <NUM>
|   <TRUE>
|   <FALSE> )
    { System.out.println("inside fragment"); }
}
```

When compiling the parser I encountered errors with left recursion and choice conflicts.
Expression() had given me two choice conflict errors.



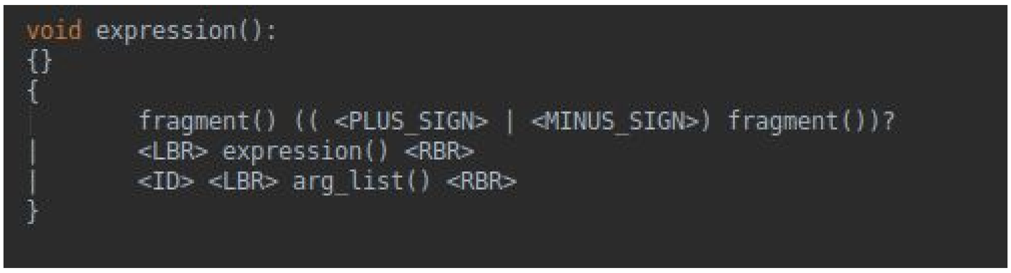
```
Warning: Choice conflict involving two expansions at
         line 269, column 9 and line 272, column 9 respectively.
         A common prefix is: "-" <ID>
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 271, column 9 and line 272, column 9 respectively.
         A common prefix is: <ID>
         Consider using a lookahead of 2 for earlier expansion.
```

I tackled the second error first. As I examined the code below, I realised that the production rule did not know which *fragment()* to enter.This is what caused the choice conflict.



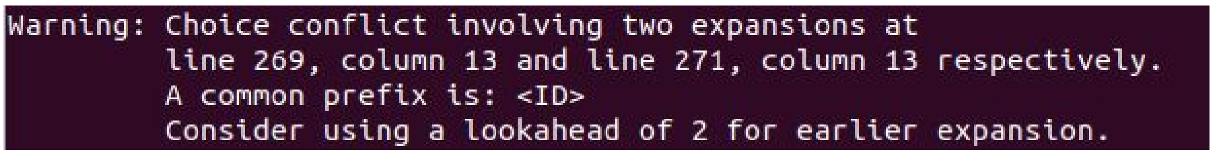
```
void expression():
{}
{
        fragment() ( <PLUS_SIGN> | <MINUS_SIGN>) fragment()
|       <LBR> expression() <RBR>
|       <ID> <LBR> arg_list() <RBR>
|       fragment()
}
```

I removed the last fragment and merged it with the first option. Wrapping the rest of the first option rule in brackets using regex's "?" made it optional:



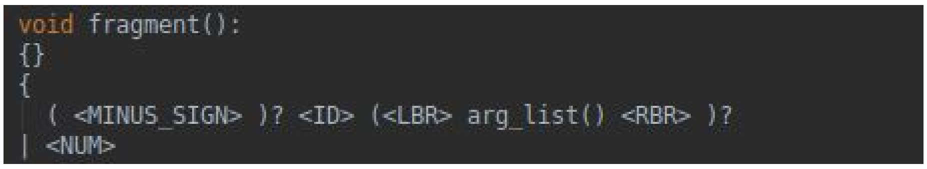
```
void expression():
{}
{
        fragment() (( <PLUS_SIGN> | <MINUS_SIGN>) fragment())?
|       <LBR> expression() <RBR>
|       <ID> <LBR> arg_list() <RBR>
}
```

Down to one error:



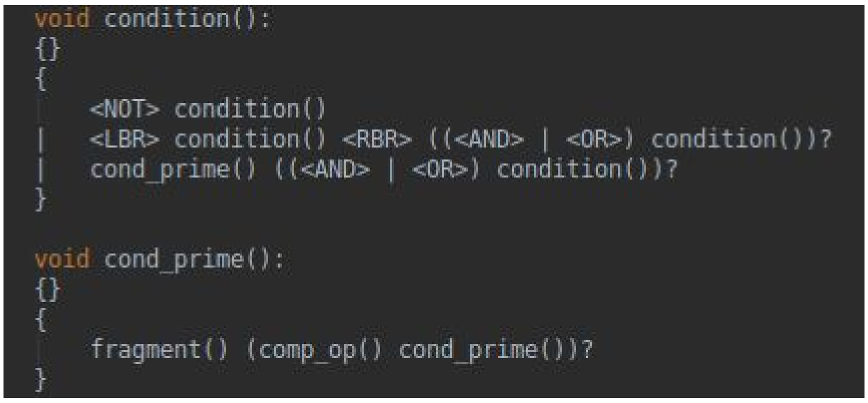
```
Warning: Choice conflict involving two expansions at
         line 269, column 13 and line 271, column 13 respectively.
         A common prefix is: <ID>
         Consider using a lookahead of 2 for earlier expansion.
```

<ID> was the common prefix to this error, this means that *<ID>* can be hit by going down more than one route. It could be hit by entering *fragment(),* means that as well as directly hitting it from *expression()*. To overcome this I removed the third option from *expression()* and made it optional inside *fragment().*



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

Left recursion was eliminated by introducing non-terminals denoted in the code with "prime" included their name.
An example is *condition(), condition_prime().*



```
void condition():
{}
{
    <NOT> condition()
|   <LBR> condition() <RBR> ((<AND> | <OR>) condition())?
|   cond_prime() ((<AND> | <OR>) condition())?
}

void cond_prime():
{}
{
    fragment() (comp_op() cond_prime())?
}
```

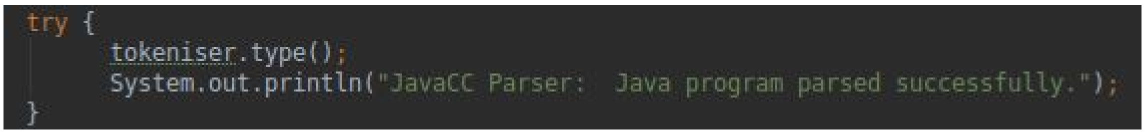
Through further testing and redesign of my parser I have renamed non-terminals as they are no longer being implemented to directly combat left recursion, but to provide abstraction to make the code easier to read and test.

This allowed for the program to run all tests correctly without using a lookahead, meaning the grammar is LL (1).

Test

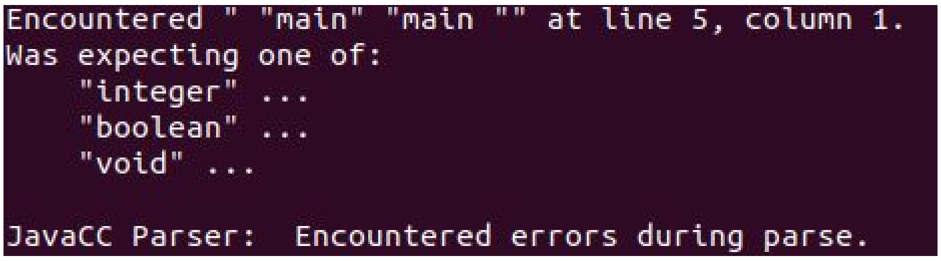
I took a test-driven approach to the implementation of the syntax analyser. From looking at the language's description I was able to design tests for each production rule. After creating each production rule I thoroughly tested it. This is why I started implementing the low level production rules. This allowed me to unit test them and I as I started to implement higher level rules I also perform integration testing.

To test each production rule I called tokenizer on that specific rule. Normally tokenizer is called on *program()* which allows it to work through the whole input file.
When I completed *type()* I called called the tokenizer on it.



```
try {
      tokeniser.type();
      System.out.println("JavaCC Parser:  Java program parsed successfully.");
}
```

My first test was a file containing the string“ integer”, This return the print statement above. I then tested it with “void” and “boolean” which both worked perfectly. I then tested it on a string “main” which returned:



```
Encountered " "main" "main "" at line 5, column 1.
Was expecting one of:
    "integer" ...
    "boolean" ...
    "void" ...

JavaCC Parser:  Encountered errors during parse.
```

This is correct as it *type()* can only evaluate to one of the three tokens created in the lexical analyser.

assignment() involved integration testing with *type().*

When testing the parser as a whole and errors occurred, having the ability to parse from a specific rule was a great advantage for debugging and finding the route of the error.

Conclusion

In conclusion I thoroughly enjoyed this assignment. At the beginning it looked very difficult, put as I broke the problem further and further down, each sub problem became achievable and manageable to integrate.

I learnt a lot about the power of regular expression and recursion. It thought the benefits of implementing recursion and the use of abstraction, making smaller functions to make the code easier to read and understand.
Completing this assignment has also given me a greater understanding of compiler construction and this module as a whole.

While completing this assignment I knew that it was important to take into consideration assignment 2, because of this I kept away from implementing kleene closures as it would not allow the AST to produce the correct node structure

Also I knew it was important to avoid Lookaheads and so this is language LL(1).

External resources

- https://javacc.org/