CA4003 – Compiler Construction Assignment 1 – A Lexical and Syntax Analyser in JavaCC

Name: Cathal Hughes

Student Number: 15417922

Stream: CASE4

The aim of this assignment was to create a lexical and syntax analyser which would accept the CAL language specified in the assignment description. We had to create a parser for the CAL language. Once this parser is initialised, it will attempt to begin parsing input using the prog() method which has been defined in the grammar. If successful it will print a message informing you, otherwise it will print a ParseException.

Whilst writing the code other errors I came across were related to left-recursion and choice conflicts. How these issues were solved are discussed in detail below.

Options

As the parser needed to be case insensitive, a specific option needed to be set to true.

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

Including this option for my parser allowed it to accept the following example:

```
MAIN
begin
eNd
```

User Code

All javacc parsers must begin with a declaration with the parser name. The name selected for my parser was “CALParser”.

Then the Parser must be initialised with a file input of the language code using the command line. If a file is not passed in via a command line-argument the user is provided with a message on how to pass in a file to the parser.

```
//Parser initialization
   CALParser parser;
   // Use console input rather than a file stream
   if(args.length == 0){
    System.out.println ("CALParser: Reading input ...");
    parser = new CALParser(System.in);
   }
   //File Stream input
   else if(args.length == 1){
    System.out.println ("" );
    System.out.println ("CALParser: Reading the file " + args[0] + " ..." );
    System.out.println ("" );
    try {
     parser = new CALParser(new java.io.FileInputStream(args[0]));
    }
    catch(java.io.FileNotFoundException e) {
     System.out.println ("CALParser: The file " + args[0] + " was not found.");
     return;
    }
   }
   else {
    System.out.println ("CALParser:  You must use one of the following:");
    System.out.println ("        java CALParser < file");
    System.out.println ("Or");
    System.out.println ("        java CALParser file");
    return ;
   }
```

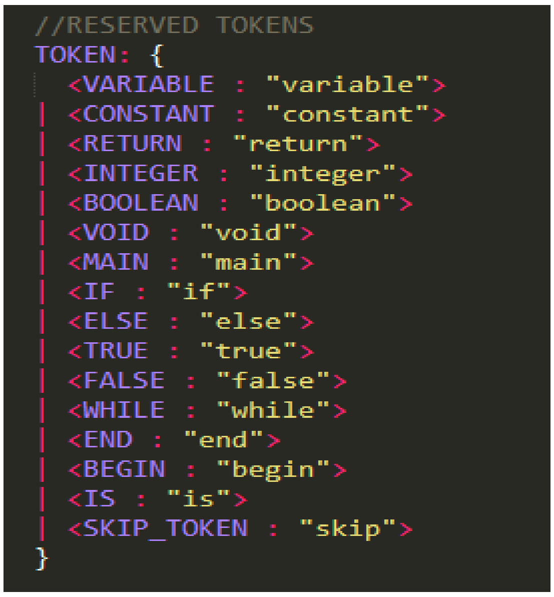
If a file is passed in the prog() method is then called, which begins the parsing of the input from the input file. If there are any errors it is here where they are thrown and they can either be a “ParseException” or a “TokenMgrError”. If the file has been parsed successfully then a message is printed saying the input has been parsed successfully.

Lexical Analysis

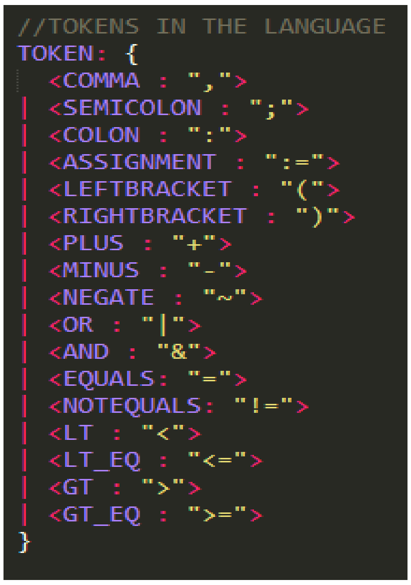
The first thing to do to complete this assignment was to perform Lexical Analysis on the input stream. In programming, a lexical analyser is the part of a compiler or a parser that break the input language into tokens. A token is the minimal meaning component. Common tokens are identifiers, integers, floats, constants, etc. All the tokens required were given in the assignment description.

Tokens

The tokens for the language consisted of reserved words and then extra tokens that were in the language. Some of the reserved words of the language included: variable, constant, return, integer, Boolean etc. Whilst some of the other tokens in the language were: comma (,), assignment (:=), Boolean operators (| &) . Here is an image of the tokens in the language:

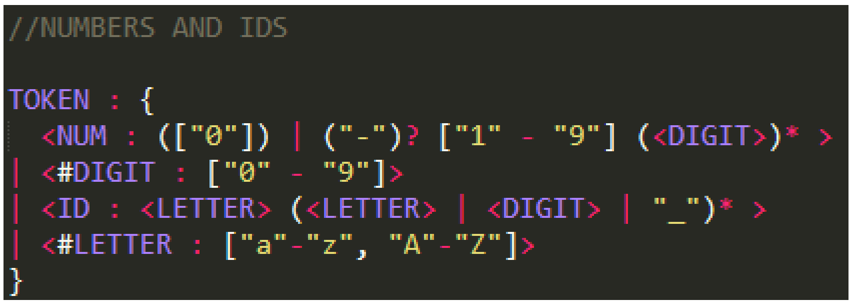


```
//RESERVED TOKENS
TOKEN: {
  <VARIABLE : "variable">
| <CONSTANT : "constant">
| <RETURN : "return">
| <INTEGER : "integer">
| <BOOLEAN : "boolean">
| <VOID : "void">
| <MAIN : "main">
| <IF : "if">
| <ELSE : "else">
| <TRUE : "true">
| <FALSE : "false">
| <WHILE : "while">
| <END : "end">
| <BEGIN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



```
//TOKENS IN THE LANGUAGE
TOKEN: {
  <COMMA : ",">
| <SEMICOLON : ";">
| <COLON : ":">
| <ASSIGNMENT : ":=">
| <LEFTBRACKET : "(">
| <RIGHTBRACKET : ")">
| <PLUS : "+">
| <MINUS : "-">
| <NEGATE : "~">
| <OR : "|">
| <AND : "&">
| <EQUALS: "=">
| <NOTEQUALS: "!=">
| <LT : "<">
| <LT_EQ : "<=">
| <GT : ">">
| <GT_EQ : ">=">
}
```

Other Tokens included Numbers, Letters, IDs and Digits. These were the most important to get correct as Numbers could be a string of digits with a minus in front of them unless it was a zero. Also, a number could not start with a leading zero. Identifiers could be represented by a string of letters, digits or an underscore but they had to start with a letter. I used Regular Expressions to solve these requirements. In the code snippet below you can see how I used regular expressions.



```
//NUMBERS AND IDS

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

The #-sign seen before DIGIT and LETTER are help definitions that do not produce tokens. Both are used in the production of tokens for ID and NUM.

It is also worth noting that javacc has its own reserved words like TOKEN and SKIP. Therefore, my token for skip was named SKIP_TOKEN.

Skip

Of course, the parser had to skip certain tokens in order to function correctly. This means it had to ignore certain characters whilst scanning for valid tokens. Characters that had to be skipped were white space characters, new line characters, carriage return characters and tab characters.

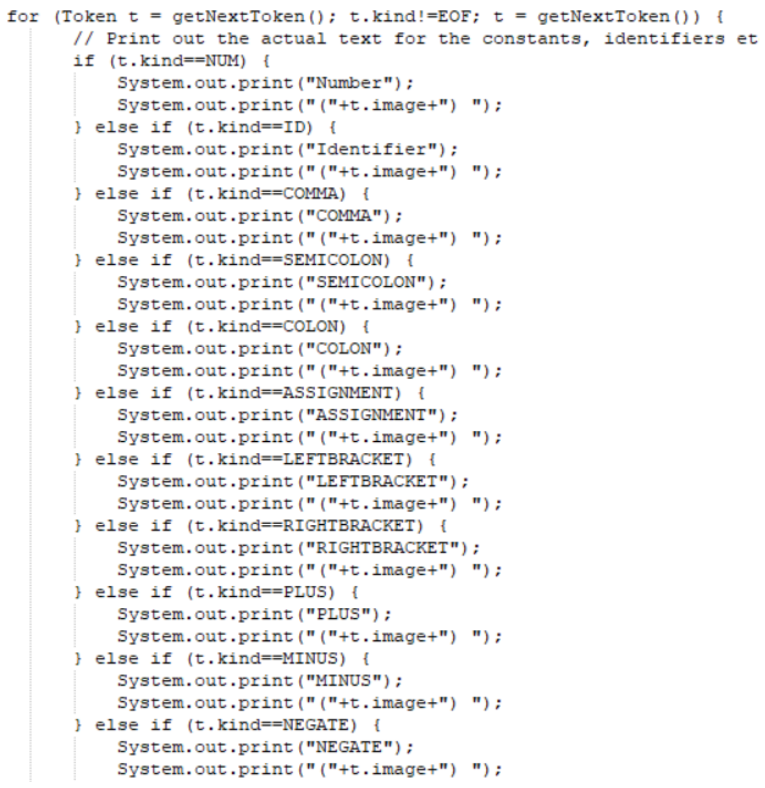
The parser had to be able to handle comments. Bothe single-line comments "//", multi-line comments "/* */" and nested comments. An example of a nest comment would be:

/∗ a comment /∗ with /∗ s e v e r a l ∗/ n e s t e d ∗/ comments ∗/

The above situation is handled by incrementing a commentNesting variable when * is encountered and changes to the IN_COMMENT state. While it is in this new state, the commentNesting variable is incremented on every /* and decremented on every */. All other tokens are skipped. When the commentNesting is returned to zero we switch back to the DEFAULT lexical state.

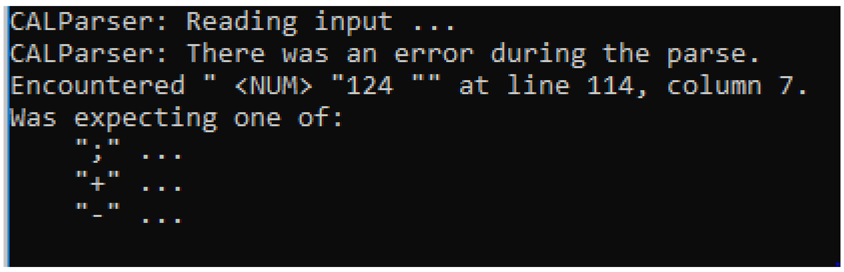
Testing Lexical Analysis

To test my code at this point I wrote code which reads in every token and prints out its kind and image. Part of this code can be seen here:



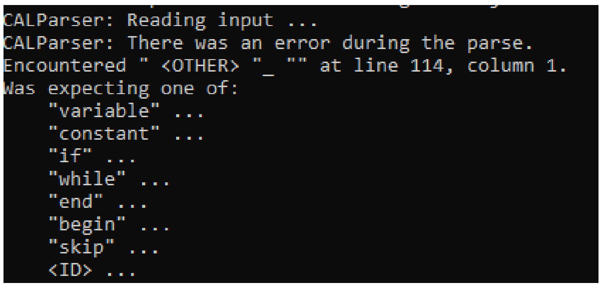
```
for (Token t = getNextToken(); t.kind!=EOF; t = getNextToken()) {
    // Print out the actual text for the constants, identifiers et
    if (t.kind==NUM) {
        System.out.print("Number");
        System.out.print("("+t.image+") ");
    } else if (t.kind==ID) {
        System.out.print("Identifier");
        System.out.print("("+t.image+") ");
    } else if (t.kind==COMMA) {
        System.out.print("COMMA");
        System.out.print("("+t.image+") ");
    } else if (t.kind==SEMICOLON) {
        System.out.print("SEMICOLON");
        System.out.print("("+t.image+") ");
    } else if (t.kind==COLON) {
        System.out.print("COLON");
        System.out.print("("+t.image+") ");
    } else if (t.kind==ASSIGNMENT) {
        System.out.print("ASSIGNMENT");
        System.out.print("("+t.image+") ");
    } else if (t.kind==LEFTBRACKET) {
        System.out.print("LEFTBRACKET");
        System.out.print("("+t.image+") ");
    } else if (t.kind==RIGHTBRACKET) {
        System.out.print("RIGHTBRACKET");
        System.out.print("("+t.image+") ");
    } else if (t.kind==PLUS) {
        System.out.print("PLUS");
        System.out.print("("+t.image+") ");
    } else if (t.kind==MINUS) {
        System.out.print("MINUS");
        System.out.print("("+t.image+") ");
    } else if (t.kind==NEGATE) {
        System.out.print("NEGATE");
        System.out.print("("+t.image+") ");
```

At this stage I wanted to make sure only the correct format for both Numbers and Identifiers were being accepted. When I used “0124” as number, this is the error that I was given:



```
CALParser: Reading input ...
CALParser: There was an error during the parse.
Encountered " <NUM> "124 "" at line 114, column 7.
Was expecting one of:
    ";" ...
    "+" ...
    "-" ...
```

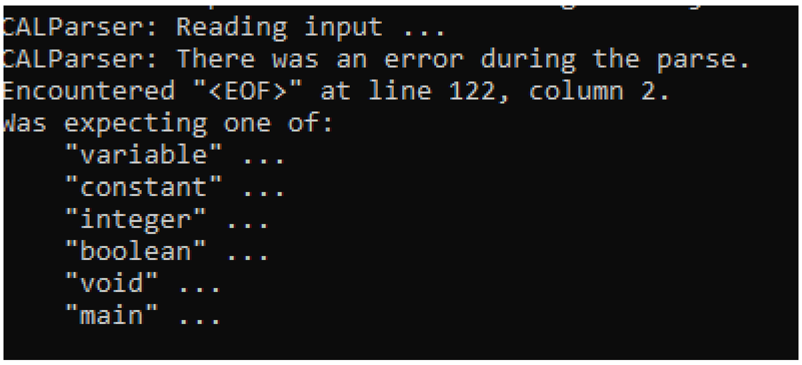
Also, when “_a” was used as an ID a similar occurred:



```
CALParser: Reading input ...
CALParser: There was an error during the parse.
Encountered " <OTHER> "_ "" at line 114, column 1.
Was expecting one of:
    "variable" ...
    "constant" ...
    "if" ...
    "while" ...
    "end" ...
    "begin" ...
    "skip" ...
    <ID> ...
```

This highlighted that only Numbers and IDs would be accepted in the correct format.

When testing the comments and nested comments I just commented out my whole file I used for testing the parser. This worked successfully too as from the image below you can see that the parser skipped everything and just read the <EOF> token.



```
CALParser: Reading input ...
CALParser: There was an error during the parse.
Encountered "<EOF>" at line 122, column 2.
Was expecting one of:
    "variable" ...
    "constant" ...
    "integer" ...
    "boolean" ...
    "void" ...
    "main" ...
```

Syntax Analysis

Defining the grammar was a methodical process which involved following the grammar specified in the assignment description. Terminals were represented by tokens whereas non-terminals were represented by method calls. The purpose of syntax analysis is to take in a stream of tokens and creating a valid sentence out of them.

The Empty String

To represent the empty string when writing the grammar I used “?”. Any time a rule should evaluate to the empty string it uses “?”. I felt this was the cleanest way to write my grammar and saved me adding the more verbose and extra “OR” case and “{}”.

Left-Recursion

I wrote the grammar as per the assignment spec and when I tried to compile the file I was greeted by some left-recursion errors. Below is an example of indirect left-recursion and how it was eliminated.

Initially my code looked like this:

```
void expression() : {}
{
        fragment() binary_arith_op() fragment() |
        <LEFTBRACKET> expression() <RIGHTBRACKET> |
        <ID> <LEFTBRACKET> arg_list() <RIGHTBRACKET> |
        fragment()
}

void fragment() : {}
{
        (<MINUS>)? <ID> (arg_list()) |
        <NUM> |
        <TRUE> |
        <FALSE> |
        expression()
}
```

This is a left-recursive grammar as expression()… -> fragment()… -> expression(). The fix can be seen in the snippet below which shows that the expression() has been removed from the left hand side of the production rule in fragment().

```
void expression() : {}
{
        fragment() expression_term() |
        <LEFTBRACKET> expression() <RIGHTBRACKET> expression_term()
}
void expression_term(): {}
{
        (binary_arith_op() expression())?
}
void fragment () : {}
{
        (<MINUS>)? <ID> (arg_list())? |
        <NUM> |
        <TRUE> |
        <FALSE>
}
```

As you can see I created the expression_term() rule which evaluates to a binary_arith_op() expression() or it can evaluate to nothing using the "?". Expression() was removed from fragment and now the expression_term() non-terminal is called in both cases in expression().

Similar occurred in my condition(), but it was a form of direct left recursion. Although, it was trickier to solve, the fix was quite similar to the one above. Removing left-recursion is important as it could potentially lead to infinite recursion and of course, javacc does not allow left-recursion.

Choice Conflicts and Left-Factoring

Once I had removed the left-recursion errors from my code, my parser compiled but this time I had a number of choice conflict warnings. Some of these were straight forward to solve whilst others took a bit more time. My goal was to pass all the tests correctly without using LOOKAHEADS, which would mean the grammar was LL(1), this is something I feel I achieved. Not using a LOOKAHEAD means my code is more efficient. According to the javacc website "The default value for a LOOKAHEAD is 1. The smaller this number, the faster the parser." Initially I was using a LOOKAHEAD(3) in my condition rule. After some careful studying of the grammar I was able to remove the use of the LOOKAHEAD. Because of this, this is the part of the code I tested the most as removing the LOOKAHEAD could mean my parser accepts something it should not. As it stands, I have yet to find a case that my parser accepts that it should not.

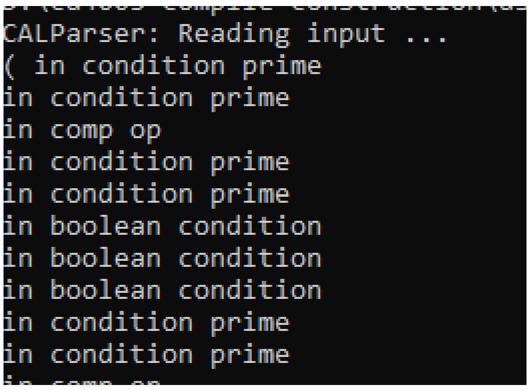
One of the easier choice conflicts to solve occurred in the statement() production rule. Initially my code looked like:

```
void statement() : {}

{

        <ID> <ASSIGNMENT> expression() <SEMICOLON>|

        <ID> <LEFTBRACKET> arg_list() <RIGHTBRACKET> <SEMICOLON> |

        <BEGIN> statement_block() <END> |

        <IF> condition() <BEGIN> statement_block() <END> <ELSE> <BEGIN> statement_block() <END> |

        <WHILE> condition() <BEGIN> statement_block() <END> |

        <SKIP_TOKEN> <SEMICOLON>

}
```

Clearly in the above code there is a choice conflict involving <ID>. To overcome this I created statement_prime() and now my code looks like so:

```
void statement() : {}
{
        <ID> statement_prime() |
        <BEGIN> statement_block() <END> |
        <IF> condition() <BEGIN> statement_block() <END> <ELSE> <BEGIN> statement_block() <END> |
        <WHILE> condition() <BEGIN> statement_block() <END> |
        <SKIP_TOKEN> <SEMICOLON>
}
void statement_prime() : {}
{
        <ASSIGNMENT> expression() <SEMICOLON> |
        <LEFTBRACKET> arg_list() <RIGHTBRACKET> <SEMICOLON>
}
```

The choice conflicts in my condition() non-terminal which took much longer to fix. The fix for the involved putting {System.out.println("in production rule")} in many places in the code. The output on my terminal now looked like this:



```
CALParser: Reading input ...
( in condition prime
in condition prime
in comp op
in condition prime
in condition prime
in boolean condition
in boolean condition
in boolean condition
in condition prime
in condition prime
```

The use of the print statements allowed me to follow the flow of the program as it was being parsed by the parser. It made it far easier to debug issues in my code as using the onscreen instructions I could follow where the parsing error occurred. Like any sort of programming, the use of print statements is essential when trying to debug the program. The use of these print statements shortened the debugging process massively.

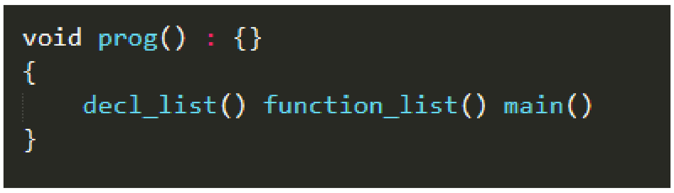
After doing this and looking at the course notes and online, I was able to apply the principals given to remove the choice conflict from the condition rule without using a LOOKAHEAD. Any new non-terminal created to avoid choice conflicts had "prime" appended to their name.

Testing Syntax Analysis

So initially I tested using the samples given in the assignment description. The multiply example makes use of most of the rules in the grammar and making sure that was passing was important.

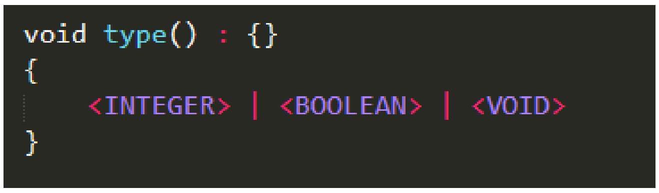
I also tested every production rule one by one by not calling parser.prog() but rather parser.statement(), parser.condition(), parser.fragment(). Doing this meant I was able to test every condition inside each of the rules individually and allowed me to test my code using my own examples rather than just using the ones provided. In some of the test cases I used I put in known errors to break my code.

It was evident that there were both high-level production rules and low-level production rules. An example of a high-level production rule was prog() as this called three other production rules.



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

An example of a low-level production rule was type() as this did not have any other calls to other production rules inside its body but strictly dealt with Tokens that had been defined in the language.



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

This meant when testing by calling each function I would start at the low-level production rules and move my way up. After testing type() I could then test the likes of var_decl() and so on. This was very useful as finding errors in low-level rules meant when I tested the high-level rules I didn't come across as may errors.

This process was very beneficial as I found some specific conditions not satisfied by the rules. In the condition rule a major issue occurred in that "(a > b) >= (c > d)" was being accepted and parsed successfully by my parser. This should not have been the case. A comparison operator should not be found between two conditions, only a Boolean logical operator like "&" or "|". Although conditions such as "(a > b) & (c > d)" and "(a > b) | (c > d)" by my parser, it was clear it needed further tweaking in order for it to match the grammar specified in the assignment. The fix for this involved ensuring every comparison operator was followed by an expression.

Overall it was the condition rule which seem to cause the most issues and proved the most difficult to implement.

Conclusion

This assignment was a nice challenge and really helped me understand some of the topics that have been covered in class. I believe my assignment will provide me with a great base for the second assignment as my solution is fully recursive and this will help me implement my Abstract Syntax Tree.

External Sources

- https://javacc.org/