A Lexical and Syntax Analyser for the CCAL Language

Table of Contents

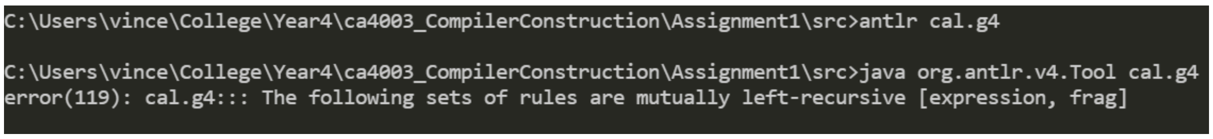
Introduction

In this report, I will discuss the core features of this assignment and describe how I went about implementing the CAL grammar file with the lexer and parser.

I first read the ccal.pdf document provided in the assignment specification to understand the language definition of CAL. I began by first breaking down the steps and expanding the grammar file throughout the implementation. I did this to better understand how the grammar file (cal.g4) would work when creating the parse tree when running it with the ANTLR command. The CAL language definition states that anything enclosed with angle brackets is a non-terminal, and anything in a **bold typeface** is a terminal. Using this information, I implemented the grammar by having the non-terminals represented by lowercase letters, and terminals represented by uppercase letters. I made some CAL files throughout the implementation of the grammar file in order to test the grammar (e.g: comments.cal, caseInsensitive1.cal, etc, all of which are located in the tests directory). I focused on the CAL syntax specified in the ccal.pdf document, continuing to convert the language definition to the appropriate grammar in cal.g4. I came across some issues during this step, such as using SKIP and fragment for the language definition (for the lexer and parser rules, respectively). These couldn't be used because they are reserved keywords defined in the ANTLR syntax. I fixed this by simply renaming SKIP with SKIPPING, and fragment with frag. The parser uses the tokens defined by the lexer rules and this allows for the parsing process.

Eliminating Indirect Left Recursion

When first implementing my grammar by carefully reading the CAL language definition from ccal.pdf, I noticed the error message regarding left-recursion in my code. In particular, it was related to how expression and fragment are mutually left-recursive and essentially means there's an infinite loop. ANTLR is a top-down parser, and left recursion doesn't work well with top-down parsers.



```
C:\Users\vince\College\Year4\ca4003_CompilerConstruction\Assignment1\src>antlr cal.g4

C:\Users\vince\College\Year4\ca4003_CompilerConstruction\Assignment1\src>java org.antlr.v4.Tool cal.g4
error(119): cal.g4::: The following sets of rules are mutually left-recursive [expression, frag]
```

Expression had indirect left recursion, which means that expression calls fragment which calls expression, etc. Hence, this can lead to an infinite loop. This resulted in me having to remove expression from the fragment rule to fix this issue. Below shows the before and after comparison of that code segment.

```
// before
frag:                      ID
                         | MINUS ID
                         | NUMBER
                         | TRUE
                         | FALSE
                         | expression
                         ;
```

```
// after (removed expression)
frag:                      ID
                         | MINUS ID
                         | NUMBER
                         | TRUE
                         | FALSE
                         ;
```

Letters and Words

In the first section of the lexer rules, I used fragments that would be used in defining keywords with uppercase and lowercase letters. This allowed the language to be case insensitive, for instance, the caseInsensitive.cal files (located in the tests directory) each have the main keyword used with different cases. Therefore, I defined rules to allow for this functionality.

```
fragment A:              ('a' | 'A');
fragment B:              ('b' | 'B');
fragment C:              ('c' | 'C');
fragment D:              ('d' | 'D');
fragment E:              ('e' | 'E');
fragment F:              ('f' | 'F');
...
```

This allows for letters to match their corresponding case, for instance, fragment A can match with ‘a’ or ‘A’, fragment B could match with ‘b’ or ‘B’, etc. Within the lexer rules, this allowed me to define the keywords with these fragments.

```
CONST:                   C O N S T;
INTEGER:                 I N T E G E R;
BOOLEAN:                 B O O L E A N;
...
```

This functionality allows for case insensitivity, as mentioned before with the caseInsensitive.cal files, the term main can be used with different cases (e.g: main, Main, MAIN, etc).

The code below shows the rule defining what a letter, digit, and underscore is. It uses regular expressions for pattern matching. For example, the letter “a” is a Letter because the rule below states a Letter can be lowercase or uppercase. This is similar for Digit, and Underscore is defined as a literal ‘_’. The single quotes, as seen in the fragments for the letters above, means that we’re matching with the literal itself.

```
fragment Letter:         [a-zA-Z];
fragment Digit:          [0-9];
fragment UnderScore:     '_';
```

Operators and Separators

My tokens definitions for the operators are similar to how they’re defined in the CAL language definition notes. Using the sample code in that document, I ensured that the correct operators and separators were used so as to not cause confusion when parsing. At first, I had the ASSIGN operator as ‘:=’ and the EQUAL operator as ‘=’, but I then changed the ASSIGN and EQUAL operators to ‘=’ and ‘==’, respectively. In my code cal.g4, I ordered the operators as listed in the CAL language definition documentation just for ease of readability when comparing it with my grammar.

```
ASSIGN:                  '=';
PLUS:                    '+';
MINUS:                   '-';
...
EQUAL:                   '==';
NOTEQUAL:                '!=';
LT:                      '<';
...
```

As for the separators, I defined them similarly to how the operators are defined. Instead of using BEGIN and END in my grammar, I decided to go with the use of { and }, respectively. This was because upon first starting my implementation, I broke down the problem into a series of steps and noticed that the examples in the CAL language definition document used braces instead of BEGIN and END tokens. This made it easier to follow for me and to allow for consistency with the CAL examples.

```
COMMA:                   ',';
SEMI:                    ';';
COLON:                   ':';
LBRACE:                  '{';
RBRACE:                 '}';
LBRACK:                 '(';
RBRACK:                 ')';
```

Integers and Identifiers

The CAL language definition document shows how the number and identifier tokens are to be interpreted. It strictly states that:

- Integers can be described as a string of one or more digits (‘0’-‘9’) that do not begin with the digit ‘0’, but they can start with a minus sign. Hence, the NUMBER rule states that a number can match with a ‘0’, or an optional MINUS sign (zero or one time depicted by the question mark symbol) followed by one digit between 1-9 and second digit (ranging from 0-9) which may occur zero or more times.
- Identifiers are to be represented by a string of letters, digits or underscore characters. Identifiers can begin with a letter, but they cannot be reserved words. In my grammar, I translated this to define an identifier that can match with a letter (lowercase or uppercase letters) followed by either another letter, a digit, or an underscore, zero or more times. Letter, Digit, and Underscore have been defined in the lexer rules and described in the report above.

```
NUMBER:                 '0' | (MINUS? [1-9] Digit*);
ID:                     Letter (Letter | Digit | UnderScore)*;
```

Whitespace and comments

The CAL language definition states that there are two types of comments: single-line and multi-line comments, represented by // and /* */ respectively. I have it implemented so the parser ignores whitespaces and comments, so the rules WS, COMMENT, and NESTED_COMMENT skip the whitespaces, comments, and nested comments. I looked into how to deal with multi-line comments [1] in order to better understand the pattern matching process for it.

```
WS:                     [ \t\n\r]+ -> skip;
COMMENT:                '//' .*? '\n' -> skip;
NESTED_COMMENT:         '/*' (NESTED_COMMENT|.)*? '*/' -> skip;
```

Epsilon

When dealing with epsilons from the CAL language definition, I had to do some research in the notes and online to figure out how to get the regular expressions for it. I encountered this StackOverflow post [2] which describes the grammar for dealing with epsilons which simply states that epsilon is matched with nothing but you can use a comment to indicate that it represents an epsilon.

```
someRule:                 A
                        | B
                        | /* epsilon */
                        ;
```

Which is as equal as:

```
rule:                     A
                        | B
                        |
                        ;
```

Since epsilons are used to match nothing, I noticed the use of the “?” operator did the same as what the above code does. “?” means that something is optional and would match zero or one time. Therefore, it is equivalent to the epsilon in this language definition. An example of this is shown below.

```
function_list |= (<function> <function_list> | ϵ)
```

This is equivalent to:

```
functionList:           (function functionList)?;
```

Error Handling

In order for me to have the java parser print out a custom message as to whether the parser ran successfully or not, I had to do some research on the ANTLRErrorListener library and figure out how to remove the error listener and make my own. This would allow me to have a custom error message after parsing the CAL files in the tests directory. At first, I wanted to keep things simple by trying to implement a try/catch in my cal.java code to print whether the parser succeeded or not. I wanted to have it so the try clause would print “<fileName.cal> parsed successfully” if no errors occurred, and in the catch clause have “<fileName.cal> did not parse successfully”. This did not work however because ANTLR generates default error listeners, so simply having the try and catch clauses in my cal.java did not work. After looking into this further [3, 4], I figured I needed to have it so the default listeners are removed and have it run my own version of the error listener by overriding it. This allowed me to have my custom error message print to stdout upon coming across an error when parsing. The code below showing the removal of the default error listeners can be found between lines 26-42 in cal.java located in the src directory.

```
lexer.removeErrorListeners();
parser.removeErrorListeners();
 // setting up the syntax error listener with our own custom error message
CalErrorListener errorHandling = new CalErrorListener(fileName);
parser.addErrorListener(errorHandling);
```

Hence, whenever a syntax error occurs, it will call the syntaxError method which prints out my custom error message as well as increments the errorCount. I used the errorCount as an indicator mechanism in order for the parser to know whether to print the successful message or not. Essentially, whenever an error occurs, the custom error message is printed, errorCount increments, and this prevents the “successfully parsed” message from being outputted from cal.java. If there are no errors when parsing, the errorCount is not incremented and will trigger the “successfully parsed” message to print in stdout.

```
if(errorHandling.errorCount == 0) {
    System.out.printf("%s parsed successfully!\n", fileName);
}
```

I also have my code to initialise the filename which is passed in as an argument when running cal.java to prevent the stdout message from showing the file path, but rather the name of the CAL file being parsed for readability purposes.

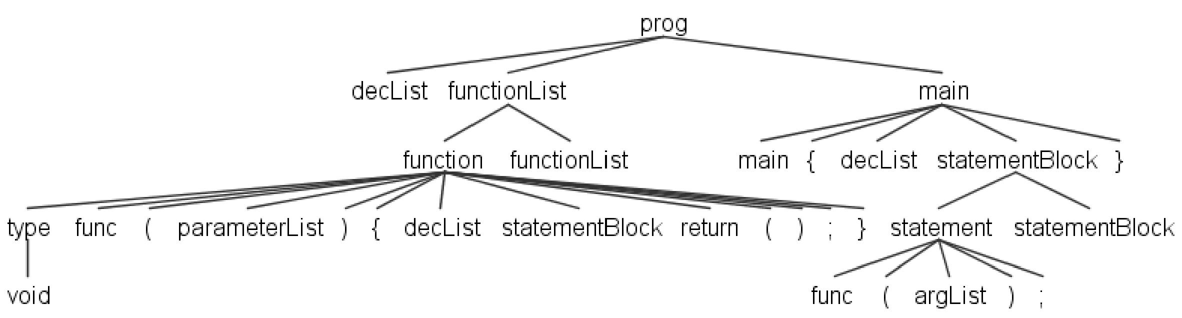
```
// getting the file name for the output message rather than having the path
included for the output
private static String getFileName(String inputFile) {
    if (inputFile == null) {
        return "input";
    }
    Path path = Paths.get(inputFile);
    return path.getFileName().toString();
}
```

Testing

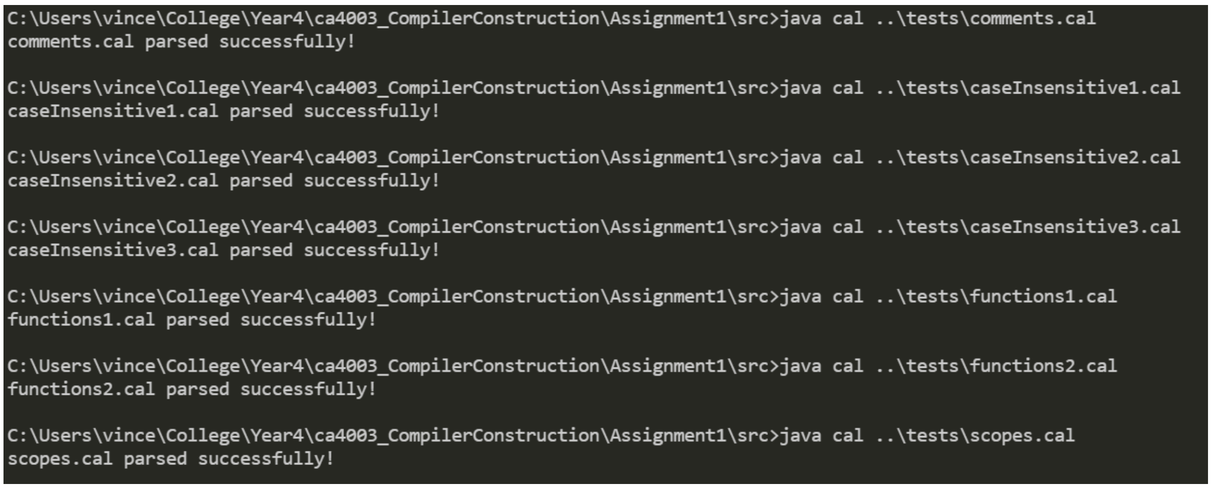
Below are the results I obtained after running the parser on each of my test CAL files (located in the tests directory), as well as an example parse tree output when running the grun batch command on functions1.cal.

This was the resulting parse tree for functions1.cal when running the following command:

```
>grun cal prog ..\tests\functions1.cal -gui
```



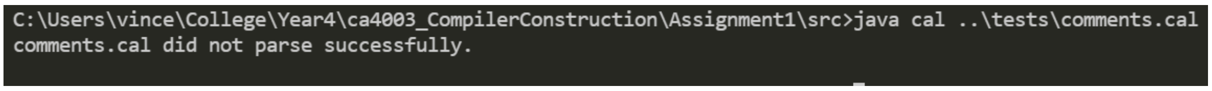
Below are the results of running the parser on each CAL file.



```
C:\Users\vince\College\Year4\ca4003_CompilerConstruction\Assignment1\src>java cal ..\tests\comments.cal
comments.cal parsed successfully!

C:\Users\vince\College\Year4\ca4003_CompilerConstruction\Assignment1\src>java cal ..\tests\caseInsensitive1.cal
caseInsensitive1.cal parsed successfully!

C:\Users\vince\College\Year4\ca4003_CompilerConstruction\Assignment1\src>java cal ..\tests\caseInsensitive2.cal
caseInsensitive2.cal parsed successfully!

C:\Users\vince\College\Year4\ca4003_CompilerConstruction\Assignment1\src>java cal ..\tests\caseInsensitive3.cal
caseInsensitive3.cal parsed successfully!

C:\Users\vince\College\Year4\ca4003_CompilerConstruction\Assignment1\src>java cal ..\tests\functions1.cal
functions1.cal parsed successfully!

C:\Users\vince\College\Year4\ca4003_CompilerConstruction\Assignment1\src>java cal ..\tests\functions2.cal
functions2.cal parsed successfully!

C:\Users\vince\College\Year4\ca4003_CompilerConstruction\Assignment1\src>java cal ..\tests\scopes.cal
scopes.cal parsed successfully!
```

If however there was a syntax error in one of the CAL files, the error message will be printed to stdout. For example, if comments.cal had improper syntax, my custom error message should be displayed when attempting to parse comments.cal.

```
// comments.cal with improper syntax
main
{knfd fdfksmsfsfw
    // a simple comment
    /* a comment /* with /* several */ nested */ comments */
}
```



```
C:\Users\vince\College\Year4\ca4003_CompilerConstruction\Assignment1\src>java cal ..\tests\comments.cal
comments.cal did not parse successfully.
```

References

[1] lischke, M. and ashley, D., 2017. *How to handle nested comments in ANTLR lexer - ANTLR4*. [online] Daily-blog.netlify.app. Available at: <https://daily-blog.netlify.app/questions/2185833/index.html> [Accessed 18 October 2021].

[2] Kiers, B. and Baxter, I., 2011. *What is the equivalent for epsilon in ANTLR BNF grammar notation?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/5522574/what-is-the-equivalent-for-epsilon-in-ANTLR-bnf-grammar-notation/5522603#5522603> [Accessed 18 October 2021].

[3] Jonkman, N., Stack Overflow, 2018. *Unable to catch ANTLR4 Syntax errors*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/53593767/unable-to-catch-ANTLR4-syntax-errors> [Accessed 18 October 2021].

[4] xNappy, Stack Overflow, 2016. *ANTLRErrorListener to get error msg*. [online] Available at: <https://stackoverflow.com/questions/35429778/ANTLRerrorlistener-to-get-error-msg> [Accessed 18 October 2021].