A Lexical and Syntax Analyser Report

Plagiarism Declaration

Name(s): Janesh Sharma
Student ID(s): 17473124
Programme: CASE4
Module Code: CA4003
Assignment Title: A Lexical and Syntax Analyser
Submission Date: 15/11/20
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 sources 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): _____Janesh Sharma_____ Date: _____14/11/20_____

Table Of Contents

Implementation

To begin implementing the Cal parser I first defined all of the lexer rules followed by the parser rules as defined in the language spec exactly. Some small issues arose due to naming conventions, for example SKIP could not be used as a lexer rule as it was defined as part of the antlr syntax. The same issue arose for the fragment parser rule, the solution of both of these was to simply rename the rules:

fragment -> frag
SKIP -> SKIP_STATMENT.

Note that SKIP is still a valid keyword to be used in an input file.cal

Case Insensitive

To make the language case insensitive I defined fragments for every letter in the alphabet e.g. :

```
// alphabet of lowercase and uppercase letters
fragment A :            [aA];
fragment B :            [bB];
fragment C :            [cC];
...
```

[aA] matches either an “a” or an “A”. I would then define my lexer rule keywords using these fragments e.g.

```
MAIN:                   M A I N;
```

This allows for main to be defined as any variation of uppercase and lowercase characters of the word main such as “mAiN”.

Epsilon

Epsilon is used to match nothing. Anywhere in the language spec that specified an epsilon symbol ε, I placed an optional operator “?” next to the rule(s) as appropriate. For example the rule defined as:

```
<decl_list> |= (<decl> ; <decl_list> | ε )
```

Becomes:

```
dec_list:               (decl SEMI dec_list)?;
```

? means it is optional, thus matching zero times or once. This is equivalent to epsilon as given an optional statement M?, M? Is equivalent to M|ε.

Error Handling

To print whether or not the file parsed successfully, I had to remove the default error listener and create my own using the Java code below. The first two lines of code removes the default error listeners implemented by antlr for the lexer and parser. This removes error messages from being written to stdout. In the third line I create my own ErrorHandler class to allow me to handle the errors myself and output a custom error message such as “file.cal has not parsed. In the last line I pass my custom ErrorHandler class as a parameter to set all parser errors to be sent there.

```
lexer.removeErrorListeners();
parser.removeErrorListeners();
ErrorHandler errorHandler = new ErrorHandler(fileName);
parser.addErrorListener(errorHandler);
```

Every time a syntax parser error occurs the syntaxError method below is called. Here I print my custom error message to stdout.

```
public class ErrorHandler implements ANTLRErrorListener {
   private String file;
   public Integer errorCount;

   public ErrorHandler(String file) {
       this.file = file;
       this.errorCount = 0;
   }

   @Override
   public void syntaxError(...) {
       System.out.println(this.file + " has not parsed");
       this.errorCount++;
   }
```

I also iterate an errorCount variable when an error occurs this is to let my cal.java file know whether or not an error has occurred.

```
if (errorHandler.errorCount <= 0)
   System.out.println(fileName + " parsed successfully");
```

Here I check if there has been an error and if not I print my message to stdout indicating parsing has been successful.

Skip Whitespace And Comments

The WS rule below skips all whitespace. The LINE_COMMENT rule skips single line comments. The MUTLI_LINE_COMMENT skips multi-line nested comments. I acquired the rules to skip these comments from Stack Overflow. [1]

```
WS:                 [ \t\n\r]+ -> skip;
LINE_COMMENT:            '//' .*? '\n' -> channel(HIDDEN);
MULTI_LINE_COMMENT:      '/*' (MULTI_LINE_COMMENT|.)*? '*/' ->
channel(HIDDEN);
```

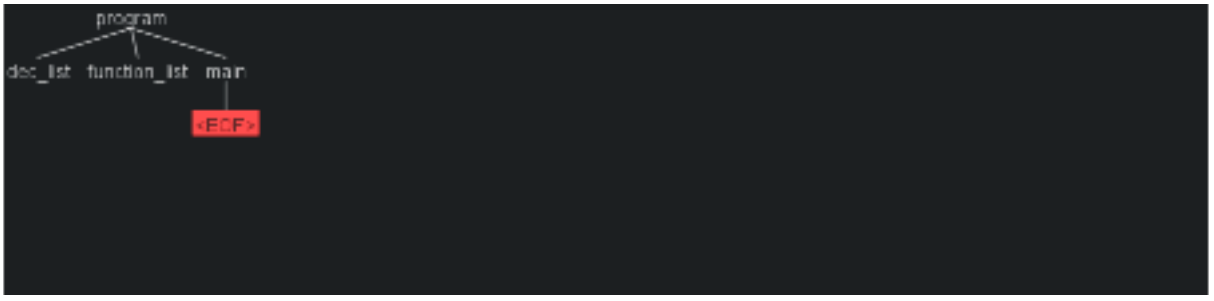
Multi-Add/Sub

I added the bracket and * symbol to the parser rule below to allow multiple additions or subtractions within an expression. For example x := a + b + c + d is now valid whereas without my addition only x := a + b would be valid.

```
expression:              frag (binary_arith_op frag)*
```

EOF Error

When the input file.cal is blank you get an error in the parse tree as shown below.



My solution to this was to wrap the main program parser rule in brackets and or it with the EOF inbuilt antlr keyword which stands for end of file, which removes the error:

```
program:                (dec_list function_list main) | EOF;
```

Number

The rule below defines a valid Number in my grammar. The regex expression below matches a ‘0’ or a number with the first digit from 1 to 9 and the second digit as a number from 0 to 9 zero or more times with an optional minus before the number.

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

Identifier

The rule below defines a valid Letter in my grammar. The regex expression matches a letter(a-ZA-Z) followed by either a letter or a digit(0-9) or an underscore zero or more times.

```
ID:                 Letter (Letter | Digit | UnderScore)*;
```

References

1. “How to Handling Nested Comments in Antlr Lexer.” Stack Overflow, stackoverflow.com/questions/27539351/how-to-handling-nested-comments-in-antlr-lexer. Accessed 14 Nov. 2020.