Declaration on Plagiarism

Assignment Submission Form

This form must be filled in and completed by the student(s)submitting an assignment

Name(s): Killian Connolly
Programme: Computer Applications & Software Engineering
Module Code: CA4003
Assignment Title: A Lexical And Syntax Analyser
Submission Date: 15th November 2020
Module Coordinator: David Sinclair

I/Wedeclare that this material, which I/Wenow submit for assessment, is entirely my/ourown 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/ourwork. I/Weunderstand 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/Wehave read and understood the Assignment Regulations. I/Wehave 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/usor any other person for assessment on this or any other course of study.

I/Wehave read and understood the referencing guidelines found at http://www.dcu.ie/info/regulations/plagiarism.shtml, https://www4.dcu.ie/students/az/plagiarismand/or recommended in the assignment guidelines.

Name(s): Killian Connolly Date: 15th November 2020

Introduction

I will briefly describe my lexical and syntax analyser, elaborating on implementation of major components. The java class will be described first followed by the main implementations and changes made to the grammar file.

CAL Class

The cal class is similar to the example in the notes on the Simple Programming Language. I have expanded on the class adding file exceptions and extending the BaseListener Class in order to output the success of the given file.

Successfully/Unsuccessfully Parsed

Obtaining syntax errors is simply calling getNumberOfSyntaxError() function on the parser. However just getting lexical errors is unachievable using this function. According to the co-founder of Antlr the best way of not having lexical errors is to make your grammar take in all possible tokens and the parser should handle the inconsistencies.
I could have completed this however I decided to abide by a more simpler approach by extending the BaseErrorListener. Having a condition that, if the parser or lexer manages to go into my extension, CALErrorListener, then either lexical or syntactic error has occurred. Thus, outputting that the file has not parsed correctly.

```
public static Boolean errorCondition = false;
```

In order for the parser not to call the default ErrorListeners, I simply had to remove them and add my extended ErrorListener, CALErrorListener.

```
lexer.removeErrorListeners();
lexer.addErrorListener(CALErrorListener.INSTANCE);
```

Explaining CAL Class

Once the lexer takes in the streams of characters while calling the modified ErrorListener class, the lexer stream is converted into tokens using the CommonTokenStream() function. These tokens are case-insensitive meaning each of these tokens are represented to the fragments in the grammar file as being upper or lower case.
These tokens are then parsed, again calling the modified ErrorListener class if an error has occurred. The parsed tokens then call the prog() function, which completes the functions explained above.

Grammar File

All tokens and production rules that are required within the assignment specification is in the grammar file.

The following are noteworthy components:

Terminals are case-insensitive (True, tRue, TRUE, true are all accepted). Terminals are defined this way by means of using fragments. Fragments are rules that are defined within the grammar file. Defining each letter of the alphabet as fragments allow for case insensitive grammar i.e. `fragment T:  't' | 'T';`

A rule such as `True: T R U E;` each letter defined within this rule would match to the fragment that would have an option of upper or lower case.
A rule such as `True: 'T R U E';` would not be case insensitive since there are quotations. The rule would only succeed if 'T R U E' is taken in.
Identifier and **whitespace** is obtained from the notes. Identifier taking a letter followed by letter/digit/underscore any number of times denoted as (*).
Line comment is anything between two forward slashes up until a newline or return denoted as (\n , \r).
Block comment being anything in between an open comment and a close comment. Nested comments needed to be accounted for as well. Hence, the nested comment calls itself if if an open and close brackets occurs within the initial block.

```
COMMENT_1:      '//' ~('\r' | '\n')* -> skip;
COMMENT_2:      Comment_open (COMMENT_2 | .)*? Comment_close -> skip;
```

Number rule takes care of leading zeros by eliminating the possibility of having a zero as the first character. Meaning any digit from 1 - 9 must be the first digit of a number in order for the file to run successfully. Number token also contains negative numbers without leading zeros and zero itself.

Epsilons are denoted as an empty option as shown below for the 'decl_list' nonterminal.

```
decl_list:      decl SEMI decl_list
                |
                ;
```

Within the 'Comment' and 'Number' rules I make use of Kleene Closure. Within the number rule, 'Digit' can occur zero or more times. For some of the other production rules I could have incorporated Kleene Closure instead of having an empty option such as the 'decl_list' example above. In doing this, it would eliminate the need for the 'decl_list' production rule and the 'decl' production rule would be formatted as shown below:

```
decl:   (var_decl | const_decl) SEMI decl*
```

In order to avoid getting a mutually left-recursive error with the production rule **fragment** and **expression** an edit needed to be made to the expression production rule. Since Antlr is an LL() language it cannot deal with mutual left-recursion beyond one production rule. So, instead I had to call expr from the same as briefly shown below.
Previously:

```
expr:           frag binary_arith_op frag
```

Now:

```
expr:           expr binary_arith_op expr
```

An explicit **End of File(EOF)** was added to the end of the 'program' production rule to make sure anything after the original given structure of the rule is not taken into account by the parser. An empty file would unsuccessfully parse since it does not follow the structure of the prog nonterminal.

```
prog:           decl list function list main EOF;
```

For **future use** of the grammar file in the second assignment, in some parts of the grammar where optionality is required, I have given groups of tokens a name.

```
| condition op=(OR | AND) condition
```

When implementing semantic analysis we can specify what to do if either OR or AND occurs.

Explanation of the statement production rule:

```
37  stm:            ID stm_choice
38                  | Begin statement_block End
39                  | If condition Begin statement_block End Else Begin statement_block End
40                  | While condition Begin statement_block End
41                  | Skip SEMI
42                  ;
```

Line 37: The ‘stm’ production rule is defined with the first rule being an identifier followed by a statement choice. Here statement choice has been factored into another production rule consisting of two options. An assignment followed by expression then a semicolon. Or left brack expression right bracket ending with a semicolon.

```
44  stm_choice:     ASSIGN expr SEMI
45                  | LBR arg_list RBR SEMI
46                  ;
```

Line 38: Second option of the rule being, the ‘begin’ terminal followed by the calling of the statement block production rule, ending with the ‘end’ terminal.
Line 39-41: Similar to Line 38.

How to run

Any file that was given in the command line will be parsed, with disregard of the file extensions. If the argument given is a non existing file then an error is thrown.
To compile and run the parser with a given file:

```
javac *.java
java cal test_1.cal
```

Conclusion

Other than the main components described above, all other production rules in the code follow the given rules required in the specifications. All examples at the bottom of the specification requirement file parsed successfully and visually the parse trees look to be in line with the grammar. I have also created sample files myself to test against the grammar along with the cal class and they seemed to be understood by the grammar and parsed correctly.