Assignment 1: A Lexical and Syntax Analyser

Fionn Hourican

DCU

CSC1098

Dr David Sinclair

Due Date *04/11/24*

Assignment 1: A Lexical and Syntax Analyser

Plagiarism Declaration

I understand that the University regards breaches of academic integrity and plagiarism as grave and serious.

I have read and understood the DCU Academic Integrity and Plagiarism Policy. I accept the penalties that may be imposed should I engage in practice or practices that breach this policy.

I have identified and included the source of all facts, ideas, opinions and viewpoints of others in the assignment references. Direct quotations, paraphrasing, discussion of ideas 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.

I declare that this material, which I now submit for assessment, is entirely my 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 work.

I have used the DCU library referencing guidelines and/or the appropriate referencing system recommended in the assignment guidelines and/or programme documentation.

Signed: Fionn Hourican

Assignment 1: A Lexical and Syntax Analyser

This assignment aims to develop a lexical and syntax analyser using Antlr4 for the simple language CAL. This language is not case-sensitive; source code is stored in files with the .cal extension.

CAL includes several reserved words such as variable, constant, return, and if. Its token set includes common operators like +, -, and comparison operators like >= and !=. The language's integer representation does not allow leading zeros, and identifiers must start with a letter, with reserved words prohibited as identifiers. The language also supports two types of comments: block comments delimited by /* and */, and line comments starting with //.

Implementation

The Grammer

The overall structure and styling are inspired by the Antr workshops we completed over Zoom. To make the language case insensitive I used the technique described in draw.g4. I followed the structure provided in the language definition file but made one small change. As seen below I removed *expression* from my fragment (frag) definition as it was causing an error of left recursion.

```
⟨expression⟩   ⊨  ⟨fragment⟩ ⟨binary_arith_op⟩ ⟨fragment⟩  |
                  ( ⟨expression⟩ )  |
                  identifier (⟨arg_list⟩ )  |

                     ⟨fragment⟩
⟨binary_arith_op⟩  ⊨  + | -
      ⟨fragment⟩   ⊨  identifier | - identifier | number | true | false |
                     ⟨expression⟩
```

```
// Expression handling with binary operations
expression:     frag binary_arith_op frag       # ExpressionStructure
               | LBR expression RBR             # ParenthesizedExpression
               | ID LBR arg_list RBR            # FunctionCallInExpression
               | frag                           # SingleExpression
               ;

// Fragment definitions for expressions
frag:   ID              # FragID
        | MINUS ID      # NegIdentifier
        | NUMBER        # Number
        | True          # True
        | False         # False
         ;
```

I was unsure how to deal with comments in Antlr so I explored the best ways to deal with comments. I found a solution for single-line comments on stack overflow and built on this to create a rule to deal with multiline comments. [1]

```
LINE_COMMENT: '//' ~[\r\n]* -> channel(HIDDEN); // Line comment
COMMENT: '/*' ( . | WS )* '*/' -> channel(HIDDEN); // Block comment
```

For The ID rules, I based them on the language definition. They may start with any letter and then include any combination of letters numbers or underscore characters. For the Number grammar rule, I also followed the language definition. A number may be a 0 or It may start with an optional ‘-’ followed by a digit 1-9 then any number from 0-9.

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

ID:          [a-zA-Z][a-zA-Z0-9_]*;
NUMBER:       '0' | '-'? [1-9][0-9]*;
```

Assignment 1: A Lexical and Syntax Analyser

The Parser

For the parser, I mainly used the files from draw.g4 and bp.g4. I changed bp and draw to cal.

To output the lines ‘file.cal parsed successfully’ and ‘file.cal has not parsed’ I had to add to these files. I originally added a simple try-and-catch block however this always showed the files were parsing successfully even when I tested with intentional Errors.

I decided to create a custom error listener, *calSyntaxErrorListener*, to track and handle syntax errors during the parsing process. The listener extends *BaseErrorListener* and overrides the *syntaxError* method to set a flag when errors are detected and print error messages with line and position information. This approach ensures that errors encountered during lexical analysis and parsing are properly captured.

```
public boolean hasErrors() {
        return hasErrors;
    }
```

To use my listener, I removed the default error listeners from both the parser and lexer and added my custom calSyntaxErrorListener. After parsing the input, I checked if any errors occurred using hasErrors(). If an error occurred It would now successfully output ‘file.cal has not parsed’

```
// Create an instance of the custom error listener
                    calSyntaxErrorListener errorListener = new
calSyntaxErrorListener();

                    // Remove default listeners and add the custom listener
                    parser.removeErrorListeners();
                    lexer.removeErrorListeners();
                    parser.addErrorListener(errorListener);
                    lexer.addErrorListener(errorListener);
```

```
                    ParseTree tree = parser.prog();

                    // Check if any errors occurred during parsing or lexical
analysis
                    if (errorListener.hasErrors()) {
                          System.out.println(args[0] + " has not parsed");
                    } else {
                          System.out.println(args[0] + " parsed successfully");
                    }
              } catch (Exception e) {
                    System.out.println(args[0] + " has not parsed");
              }
```

References

1. Suds, T. (2012). Catching (and keeping) all comments with ANTLR. [Online]. stackoverflow. Last Updated: 19 September 2012. Available at: https://stackoverflow.com/questions/12485132/catching-and-keeping-all-comments-with-antlr [Accessed 22 October 2024].