

**TCP2451 - Programming Language Translation**  
  
Mini Project  
*Owl’s Language Compiler*

TRIMESTER 1, 2021/2022

Prepared by:

|  |  |  |
| --- | --- | --- |
| Student Name | : | Ong Shuoh Chwen |
| Student ID | : | 1171102212 |
| Lecture Section | : | TC1V |
| Tutorial Section | : | TT1V |

Submitted to:

Mr. Nathar Shah Bin Packier Mohammad  
Multimedia University, Cyberjaya

**Table of Figures**

[Figure 1: File Structure 1](#_Toc88157779)

[Figure 2: Method in Lexer.ts - lexer() 3](#_Toc88157780)

[Figure 3: DFA 4](#_Toc88157781)

[Figure 4: Methods in Parser.ts – validateToken() & preprocessparserStack() 5](#_Toc88157782)

[Figure 5: Method in Parser.ts - readStack() 5](#_Toc88157783)

[Figure 6: Method in Parser.ts - readSingle() 6](#_Toc88157784)

[Figure 7: Method in Parser.ts - readMultiple() 6](#_Toc88157785)

[Figure 8: Method in Parser.ts - scanUpcomingInput() 7](#_Toc88157786)

[Figure 9: Method in Parser.ts - isOwlBarking() 8](#_Toc88157787)

[Figure 10: Method in Parser.ts - scanTokenStack() 8](#_Toc88157788)

[Figure 11: Method in Parser.ts - reduceShiftOnSingle() 8](#_Toc88157789)

[Figure 12: Method in Parser.ts - reduceShiftOnKeywordDouble() 9](#_Toc88157790)

[Figure 13: Method in Parser.ts - stackOnShift() 9](#_Toc88157791)

[Figure 14: Method in Parser.ts - stackOnComplete() 9](#_Toc88157792)

[Figure 15: Prompt upon input 10](#_Toc88157793)

[Figure 16: Output of the program 10](#_Toc88157794)

[Figure 17: Hybrid owl action as input 11](#_Toc88157795)

[Figure 18: Normal owl action as input 12](#_Toc88157796)

[Figure 19: Non-owl action as input 13](#_Toc88157797)

[Figure 20: Invalid input 13](#_Toc88157798)

[Figure 21: Method in Prompter.ts - run() 14](#_Toc88157799)

[Figure 22: When the user attempt to proceed with the program without inserting anything 14](#_Toc88157800)

[Figure 23: Method in Runner.ts - runParser() 15](#_Toc88157801)

**Table of Contents**

[**1.0 Introduction** 1](#_Toc88157802)

[**2.0 Designing Owl’s Language** 2](#_Toc88157803)

[**3.0 Lexical Analysis** 3](#_Toc88157804)

[**3.1 Introduction** 3](#_Toc88157805)

[**3.2 Regular Expression** 4](#_Toc88157806)

[**3.3 DFA** 4](#_Toc88157807)

[**4.0 Syntax Analysis** 5](#_Toc88157808)

[**4.1 Introduction** 5](#_Toc88157809)

[**5.0 Result** 10](#_Toc88157810)

[**5.1 Input & Output** 10](#_Toc88157811)

[**5.2 Unit Testing** 11](#_Toc88157812)

[**6.0 Node.js Packages** 14](#_Toc88157813)

[**6.1 Prompts** 14](#_Toc88157814)

[**6.2 Console Table Printer** 15](#_Toc88157815)

# **1.0 Introduction**

The entire mini project is written in TypeScript programming language, in addition with extra packages installed using Node.js. To run the program, type ***npm start*** or ***node index.js*** in the terminal to proceed.   
  
The file structure of the program is stated as below.

|  |  |
| --- | --- |
| Figure 1: File Structure | The main file to be executed is ***index.js***. Since TypeScript is a subset of JavaScript, for each TypeScript file, a JavaScript file will be generated correspondingly.   In the private folder, there are a total of 7 modules, ranging from core modules to utility modules.   ***Action.ts/Action.js***: The utility module that contains all of the actions and non-terminal’s display name.  ***Interfaces.ts/Interfaces.js***: The utility module that contains all of the interfaces used within the project. ***Lexer.ts/Lexer.js***: The core module that performs lexical analysis. ***Parser.ts/Parser.js***: The core module that performs syntax analysis. ***Prompter.ts/Prompter.js***: The utility module that prompts input from the user in command line interface (CLI). ***Runner.ts/Runner.js***: The core module that combines all necessary methods to be executed and displayed from ***Lexer.ts*** and ***Parser.ts***. ***Token.ts/Token.js***: The utility module that contains the owl’s action and it’s corresponding non-terminal symbols. |

# **2.0 Designing Owl’s Language**

The compiler language is designed using context-free grammar and written is Backnus-Naur Form (BNF).   
  
Start Symbols:

* <KEYWORD>
* <OWL\_HOOT>
* <OWL\_BARK>
* <OWL\_WHISTLE>

Terminal Symbols: {hoot, hu, woo}  
Non-terminal Symbols:

1. <KEYWORD> => hoot | hu | woo
2. <OWL\_HOOT> => hoot hoot hu <KEYWORD>
3. <OWL\_BARK> => hu hoot <KEYWORD> hoot
4. <OWL\_WHISTLE> => hu woo woo hoot <KEYWORD>

Production Rules:

1. <EXP> => <KEYWORD>
2. <EXP> => <EXP>
3. <EXP> => <EXP> <EXP>
4. <EXP> => <OWL\_HOOT> | <OWL\_BARK> | <OWL\_WHISTLE>

# **3.0 Lexical Analysis**

## **3.1 Introduction**

The source code for lexical analysis is located in ***Lexer.ts***. Below is the main function in Lexer.ts which perform validation from the user input. There are two arrays, *preRegex* and *postRegex* which stored input before and after processed by regular expression. Since lexical analysis were to display out the error of the input, thus the *preRegex* array is used, while *postRegex* array will be pass down for syntax analysis usage.   
  
 Within the for loop, the program will validate if the input are valid terminal symbols, which are hoot, hu and woo.

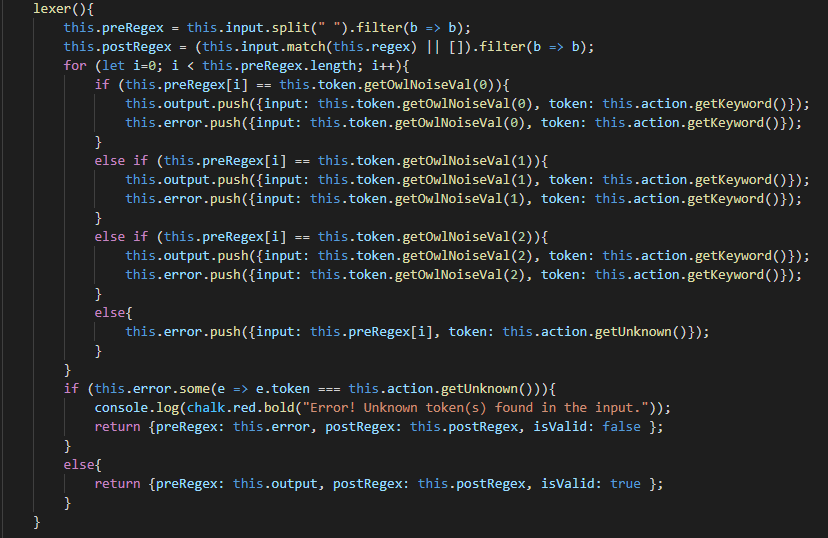


Figure 2: Method in Lexer.ts - lexer()

## **3.2 Regular Expression**

The regular expression used for lexical analysis is as below. Using this regular expression, only hoot, woo and hu with or without whitespace are valid inputs.

/((?<![\w\d])(hoot|woo|hu)(?![\w\d]))\*([/ /g])\*/gmi

|  |  |  |
| --- | --- | --- |
| **Regular Expression** | **Type** | **Description** |
| /…/ | - | An indication for JavaScript that this string is a regular expression. |
| | | Quantifier | OR in Boolean. |
| \* | Quantifier | Matches the preceding item 0 to *n* times. |
| ? | Quantifier | Matches the preceding item 0 or 1 times. |
| \w | Character Class | Find for any word character. |
| \d | Character Class | Find for any digit character. |
| ?<! | Assertion | Match only if the following pattern is preceded by predefined pattern. |
| ?! | Assertion | Match only if the following pattern is not preceded by predefined pattern. |
| (?<![\w\d]) | Pattern | Match only if the front part of the input has any word or digit characters. |
| (hoot|woo|hu) | Pattern | Match hoot, hu or woo only. |
| (?![\w\d])) | Pattern | Match only if the end part of the input has any word or digit characters. |
| ([/ /g]) | Pattern | Match globally only if there’s whitespace between inputs. |
| gmi | Flag | Flags that will affect the search. g – looks for all matches m – multiline mode  i – case-insensitive |

## **3.3 DFA**

The image below is DFA for the terminal symbols/tokens.

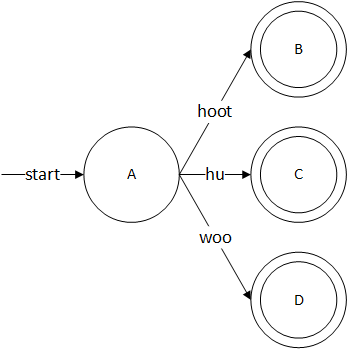


Figure : DFA

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| DFA State | hoot | hu | woo | Type |
| A | B | C | D |  |
| B |  |  |  | Accept |
| C |  |  |  | Accept |
| D |  |  |  | Accept |

# **4.0 Syntax Analysis**

## **4.1 Introduction**

The source code for syntax analysis is located in ***Parser.ts***. The syntax analysis is performed using bottom-up parsing, also known as shift-reduce technique. Below are some of the methods used to perform syntax analysis.

|  |  |
| --- | --- |
| Figure 4: Methods in Parser.ts – validateToken() & preprocessparserStack() | |
| These methods above are to validate the state of the input from lexical analysis. In ***validateToken***, if the state from lexical analysis is true, then only ***preprocessparserStack*** will be execute. While the syntax analysis is not complete, ***readStack*** will always be executed. | |
| Figure : Method in Parser.ts - readStack() | |
| The method above will be called whenever the syntax analysis is not complete. Upon initialization, an empty stack will be pushed, and proceed to verify if the upcoming token are valid owl’s actions. After initialization, based on each iteration process on the stack, check if the stack contains one value or more, so that execute each corresponding method on the run. | |
| Figure 6: Method in Parser.ts - readSingle() |
| The method above will be called whenever the stack has only one value. The stack will most likely to contain either <EXP>, hoot, hu or woo in order to trigger this method. |
| Figure 7: Method in Parser.ts - readMultiple() |
| The method above will be called whenever the stack has more than one value. |

|  |
| --- |
| Figure 8: Method in Parser.ts - scanUpcomingInput() |
| The method above is to validate if upcoming inputs are valid owl actions. If it is a valid owl action, run corresponding method (***isOwlWhistling***/***isOwlHooting***/***isOwlBarking***). Else, proceed with normal reducing and shifting based on the stack’s length (***readSingle***/***readMultiple***). |
| Figure 9: Method in Parser.ts - isOwlBarking() |
| The method above is to validate if owl is barking right now. The similar methods are ***isOwlHooting*** and ***isOwlWhistling***. All of these methods will be called within ***readSingle*** and ***readMultiple*** if the ***scanUpcomingInput*** method is true. Whenever this method is being called, the total amount will be saved and accumulated for displaying purpose later. |
| Figure 10: Method in Parser.ts - scanTokenStack() |
| The method above is to scan if the current stack still has any <KEYWORD> left that hasn’t been reduce to <EXP>. |
| Figure 11: Method in Parser.ts - reduceShiftOnSingle() |
| The method above will be triggered whenever the upcoming input are invalid owl actions, or only when the stack has exactly one value. |
| Figure 12: Method in Parser.ts - reduceShiftOnKeywordDouble() |
| The method above is to reduce <EXP> => <EXP> <EXP> or <EXP> => <EXP> <KEYWORD> into <EXP> => <EXP> and proceed with next action. |
| Figure 13: Method in Parser.ts - stackOnShift() |
| This method above is to execute basic shifting on the stack. |
| Figure 14: Method in Parser.ts - stackOnComplete() |
| This method above will only be executed once. Whenever the stack has no more <KEYWORD>(s) to be reduce and input has no more value to be pushed into the stack, *isCompleted* will be true. |

# **5.0 Result**

## **5.1 Input & Output**

|  |
| --- |
| Figure 15: Prompt upon input |
| To run the program, simply type in ***npm start*** or ***node index.js***. After that, the user would need to input the terminal symbols (hoot/hu/woo). |
| Figure 16: Output of the program |
| After the user inserted the terminal symbols, the compiler will then perform lexical analysis and syntax analysis, thus displaying the end result. **5.2 Unit Testing**  |  | | --- | | Figure : Hybrid owl action as input | | The image above shows that owl’s action can be in hybrid form, executing up to maximum of 3 actions in a row. | |
| Figure 18: Normal owl action as input |
| The image above shows that owl’s action can be a single action as well. |
| Figure 19: Non-owl action as input |
| If the input is an invalid owl action, the program will display *talking normally* instead of its corresponding owl action. |
| Figure 20: Invalid input |
| Upon invalid input, the program will display error message with indication to which of the inputs are invalid. If there’s any error in lexical analysis, syntax analysis will not be executed. |

# **6.0 Node.js Packages**

## **6.1 Prompts**

This package is used to prompt the user to input terminal symbol within the command line interface (CLI). The source code can be found in ***Prompter.ts***.

|  |
| --- |
| Figure 21: Method in Prompter.ts - run() |
| The method above is to prompt user input. Upon successful input, the method will then execute lexical analysis and syntax analysis. |
| Figure 22: When the user attempt to proceed with the program without inserting anything |
| Else, the prompt will force the user to insert value until a value has been inserted. |

## **6.2 Console Table Printer**

Even though JavaScript has built in console.table(), but this package provides advance enhancement to existing console.table(). This package is used to display syntax analysis in a prettier tabular format. The source code can be found in ***Runner.ts***.

|  |
| --- |
| Figure 23: Method in Runner.ts - runParser() |
| The method above is to display out the syntax analysis from ***Parser.ts***. Using the array list from ***Parser.ts***, the **Console Table Printer** will display the output accordingly. To highlight the output even more, chalk, a Node.js built-in library is used to differentiate different actions. For instance, red color for shifting and blue/magenta colors for reducing. |