CSM 3202: Compiler Lab

Lab Report 4:Lexer and Parser Implementation Using ANTLR4 and Flex–Bison

Name: MD. TAMIM AHMED FAHIM

ID: 2209024

Level: 3, Semester: 2

Bioinformatics Engineering

Bangladesh Agricultural University

Report submitted: January 29,2025

This is a documentation to the Lab 4 done in the Compiler Lab Course. The report is submitted to Md. Saif Uddin, Lecturer, Department of Computer Science and Mathematics, Bangladesh Agricultural University.

Contents

ListofFigures

ListofTables

Listings

1 Introduction

This lab assignment focuses on the construction and use of parsers for context-free grammars.

2 Tasks and Implementations

2.1 Task 1:Lexer and Parser Using ANTLR4

2.1.1 Objective

The objective of this experiment is to generate a lexer and parser using ANTLR4 for a given context-free grammar. The parser analyzes an input string to determine whether it can be generated by the grammar. Additionally, the parse tree of a valid input string is visualized to understand the hierarchical syntactic structure.

2.1.2 Requirements

- Java Development Kit (JDK 8 or above),
- ANTLR4 tool,
- Recursive functions corresponding to each non-terminal,
- ANTLR4 runtime library,
- Visual Studio Code.

2.1.3 Installation and Set-up

- Install Java JDK and configure environment variables.,
- Download the ANTLR4 jar file.,
- Set up ANTLR command alias for execution,
- Create a grammar file (.g4) defining lexer and parser rules.
- Generate lexer and parser files using ANTLR4 commands.

2.1.4 Implementation with GitHub Link

The grammar rules are implemented using ANTLR4. The generated lexer tokenizes the input string, and the parser verifies whether the input conforms to the grammar. For valid input, a parse tree is produced and visualized.

Click here

2.1.5 Implementation

The grammar selected for this task is:

$$S \rightarrow SWITCH\ (\ E\ )\ \{\ B\ \}$$
$$B \rightarrow L\ S^*\ D^?$$
$$L \rightarrow CASE\ E :| L\ CASE\ E :$$
$$D \rightarrow DEFAULT : S^*$$
$$E \rightarrow ID \mid NUM \mid E + E \mid E - E$$

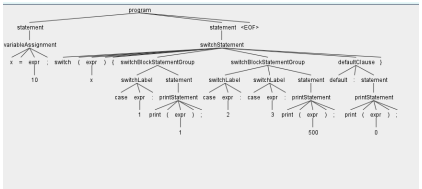
2.1.6 Input & Output

Input:

```
x = 10;

switch (x) {
    case 1:
        print(1);
    case 2:
    case 3:
        print(500);
    default:
        print(0);
}
```

Output:



2.1.7 Working Principles

- The lexer breaks the input into tokens based on defined lexical rules.,
- The parser checks the syntax using grammar rules.
- If the input satisfies the grammar, parsing is successful and a parse tree is generated.
- Invalid input produces syntax errors.

2.2 Task 2:Context-Free Grammar Parser Using Flex and Bison

2.2.1 Objective

The objective of this task is to design a lexer using Flex and a parser using Bison for a context-free grammar. The program checks whether an input string is syntactically correct and reports successful parsing or syntax errors.

2.2.2 Requirements

- Flex,
- Bison,
- GCC compiler,
- VS Code.

2.2.3 Installation and Set-up

- Install Flex and Bison using package manager ,
- Write the lexical rules in a .l file using Flex.
- Write grammar rules in a .y file using Bison.
- Compile generated files using GCC. Execute the parser program.

2.2.4 Implementation with GitHub Link

The grammar rules are implemented using Bison and flex. The generated lexer tokenizes the input string, and the parser verifies whether the input conforms to the grammar. For valid input, result is shown as 'Success: Input matches the Switch-Case grammar!'.

Click here

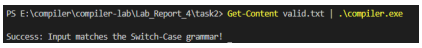
2.2.5 Input & Output

Input:

```
x = 5;
switch (x + 10) {
    case 1:
```

```
        print(1);
    case 2:
    case 3:
        {
            y = 20;
            print(y);
        }
    default:
        print(0);
}
```

Output:



2.2.6 Working Principles

- The 3-Step Process
- Lexer (Flex): Scans raw text and breaks it into tokens (e.g., converts the word switch into a code the computer understands as a ”SWITCH” category).
- Parser (Bison): Takes those tokens and checks if they follow your grammar rules (e.g., ensuring a switch is followed by (and)).
- Compiler (GCC): Glues the Lexer and Parser code together into a single .exe file that you can run.

3 Conclusion

In this lab, we implemented and analyzed different parsing techniques for context-free grammars. We designed a recursive descent parser, computed FIRST() and FOLLOW() sets, verified whether the grammar satisfies LL(1) conditions, and finally implemented a stack-based predictive parser using an LL(1) parsing table. These tasks demonstrated how syntactic analysis validates the structural correctness of input strings and ensures deterministic parsing decisions. This exercise highlights the importance of parsing in compiler design, as it bridges lexical analysis and semantic analysis by confirming that program constructs follow grammatical rules and are suitable for further compilation stages..