KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

KHULNA-9203

Project Report

Course No : CSE 3212

Course Title : Compiler Design Laboratory

Project Name : Designing A Custom Programming Language Like C

Submitted To

1) Nazia Jahan Khan Chowdhury Madam
Assistant Professor, Department of CSE, KUET

2) Dipannita Biswas Madam
Lecturer, Department of CSE, KUET

Submitted By

Md Mushfiqur Rahman
Roll: **2007027**

Section: A

Year: 3rd

Semester: 2nd

Date of Submission: 14/01/2025

Objectives:

1) To develop a custom programming language that effectively demonstrates the application of lexical analysis using Flex.
2) To utilize Bison for implementing syntax analysis and parsing in the custom language.
3) To gain hands-on experience in designing and constructing a lexer for tokenizing the input program.
4) To understand and apply context-free grammar rules for creating a robust parser.
5) To explore the integration of Flex and Bison in developing a complete compilation process.
6) To create a custom programming language that supports basic arithmetic and logical operations.
7) To implement error handling mechanisms to manage syntax and lexical errors gracefully.
8) To analyze the efficiency and performance of the custom programming language during execution.
9) To enhance problem-solving skills through the development of a compiler for a new language.
10) To document the process of building a custom language for educational and future reference purposes.

Introduction:

The project focuses on the design and implementation of a custom programming language using Flex and Bison, essential tools in the field of compiler design. Flex, a lexical analyzer generator, is utilized to tokenize the input source code by identifying keywords, operators, and other symbols. Bison, a parser generator, is employed to construct a syntax tree based on predefined grammatical rules, ensuring the code adheres to the language's syntax. This project not only demonstrates the practical application of compiler theory but also provides a comprehensive understanding of the compilation process, from lexical analysis to syntax parsing. By creating a language that supports basic arithmetic and logical operations, and incorporating robust error handling mechanisms, this project showcases the intricate interplay between lexical and syntactical components in a compiler. The experience gained through this project enhances

problem-solving abilities and offers a deeper insight into the development and optimization of programming languages.

Description:

Flex (Fast Lexical Analyzer Generator) and **Bison** (GNU Parser Generator) are powerful tools used for developing compilers and interpreters. Their roles in the compilation process are as follows:

Flex (Lexical Analysis)

- **Input**: A file containing regular expressions that define the lexical structure of the programming language.
- **Output**: A C program that scans input text and breaks it into tokens. Each token represents a meaningful sequence of characters like keywords, operators, or literals.
- **Working Mechanism**:
    - Flex reads the input file and generates a C program (typically lex.yy.c).
    - This generated program uses the defined patterns to tokenize the input source code.
    - Each match triggers a corresponding action, such as returning a token to the parser.

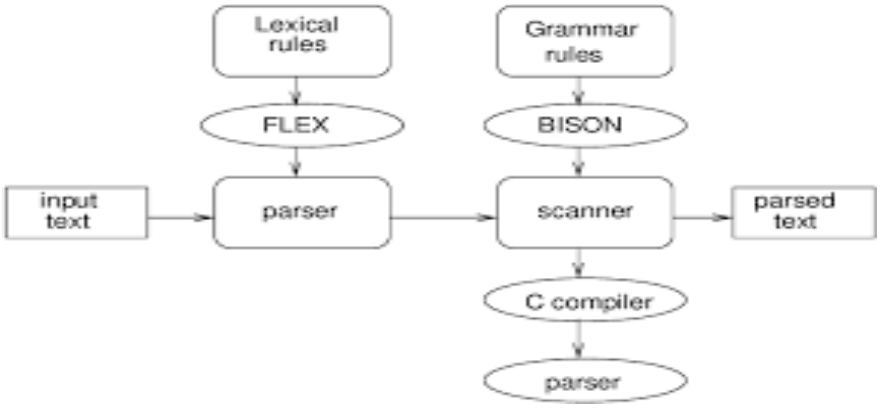

Figure 1: Working Mechanism of Flex and Bison

Bison (Syntax Analysis)

- **Input**: A file containing a context-free grammar (CFG) defining the syntactic structure of the language.
- **Output**: A C program that parses tokens produced by the lexer and checks if they conform to the grammar.
- **Working Mechanism**:
    - Bison reads the CFG and generates a parser (usually y.tab.c or mushfiq027.tab.c).
    - This parser processes the sequence of tokens, constructs a parse tree, and can execute semantic actions embedded in the grammar.
    - Bison ensures that the input follows the syntactical rules, and any deviation results in syntax errors.

An Overview of My Custom Programming Language:

My custom programming language introduces unique syntax and features designed to simplify common programming tasks while incorporating familiar constructs like loops, conditionals, and functions. Below are the key elements of the language:

Tokens:

(1) Keywords:

- **MAIN ("function main()")**: Marks the start of the main function in the program.
- **DIRECTIVE ("directive")**: Indicates a directive statement, perhaps for including files or defining constants.
- **START ("start")**: Marks the beginning of a block or section in the program.
- **END ("end")**: Denotes the end of a block or section.
- **RETURN ("rtn")**: Represents the return statement in functions.
- **IF ("iff")**: Used for conditional statements.

- **ELIF ("elif")**: Represents an "else if" condition.
- **ELSE ("else")**: Represents the else clause in conditional statements.
- **SWITCH ("switch")**: Denotes a switch case structure.
- **CASE ("case")**: Represents a case in a switch structure.
- **DEF ("default")**: Default case in a switch structure.
- **BREAK ("break")**: Used to break out of loops or switch cases.
- **FOR ("for")**: Represents the start of a for loop.
- **WHILE ("while")**: Represents the start of a while loop.
- **CIN ("take")**: Indicates input operation.
- **Function ("func")**: Denotes function declaration.
- **INT ("integer")**: Declares an integer type variable.
- **FLOAT ("float")**: Declares a float type variable.
- **CHAR ("character")**: Declares a character type variable.
- **DOUBLE ("double")**: Declares a double type variable.
- **SHOW ("show")**: Represents output operation.
- **COND ("condition")**: Possibly denotes a condition statement or block.
- **INCREMENT ("inc")**: Represents an increment operation.
- **DECREMENT ("dec")**: Represents a decrement operation.
- **GT ("gt")**: Denotes greater than comparison.
- **LT ("lt")**: Denotes less than comparison.
- **EQ ("eq")**: Represents equality comparison.
- **NEQ ("neq")**: Represents inequality comparison.
- **SQRT ("squaroot")**: Represents the square root operation.

(2) Operators and Symbols:

- **Arithmetic Operators**: +, -, *, /
- **Assignment and Comparison**: =, ->, gt, lt, eq, neq
- **Punctuation and Delimiters**: ;, ,, (,), {, }, #, <, >, <<, >>
- **Special Symbols**: @, :

(3) Comments:

- **"**-"[^*]+"-**"**: This pattern captures comments enclosed between **- and -**.

(4) Literals:

- **NUMBERI**: Matches integer numbers, possibly with a leading sign.
- **NUMBERF**: Matches floating-point numbers, with or without a leading sign.
- **VARIABLE**: Matches variable names starting with a letter, followed by letters or digits.
- **DIR**: Matches directory names (e.g., input.txt).
- **Str**: Matches string literals enclosed in double quotes (").

(5) Special Functions:

- **Mathematical Functions**: sin, cos, tan, log, power, gcd
- **Special Keywords**: evenodd, factorial, prime

Context-Free Grammar (CFG):

Program : MainBlock

MainBlock : MAIN Block

```
Block          : START StatementList END

StatementList  : Statement
               | StatementList Statement

Statement      : VariableDeclaration
               | FunctionDeclaration
               | ConditionalStatement
               | LoopStatement
               | InputStatement
               | OutputStatement
               | ExpressionStatement
               | ReturnStatement

VariableDeclaration : TypeSpecifier VARIABLE ';'

TypeSpecifier  : INT
               | FLOAT
               | CHAR
               | DOUBLE

FunctionDeclaration : "func" VARIABLE '(' ParameterList ')' Block

ParameterList  : /* empty */
               | ParameterList ',' TypeSpecifier VARIABLE
```

```
ConditionalStatement : IF '(' Expression ')' Block
                     | IF '(' Expression ')' Block ELSE Block
                     | IF '(' Expression ')' Block ElifList ELSE Block

ElifList       : ELIF '(' Expression ')' Block
               | ElifList ELIF '(' Expression ')' Block

LoopStatement  : FOR '(' Expression ';' Expression ';' Expression ')' Block
               | WHILE '(' Expression ')' Block

InputStatement : CIN VARIABLE ';'

OutputStatement: SHOW Expression ';'

ExpressionStatement : Expression ';'

ReturnStatement     : RETURN Expression ';'

Expression      : Expression '+' Expression
                | Expression '-' Expression
                | Expression '*' Expression
                | Expression '/' Expression
                | '(' Expression ')'
                | NUMBERI
```

| NUMBERF

| VARIABLE

Explanation of CFG:

Program: The root of the program, starting with the main block.

MainBlock: Consists of the keyword MAIN followed by a Block.

Block: A section of code enclosed by START and END keywords, containing a list of statements.

StatementList: A sequence of statements.

Statement: Represents various possible statements like variable declarations, function declarations, conditionals, loops, input/output, expressions, and return statements.

VariableDeclaration: Defines a variable with a specific type.

TypeSpecifier: Specifies the type of a variable (int, float, char, double).

FunctionDeclaration: Declares a function with a name, parameters, and a block of code.

ParameterList: A list of parameters in a function declaration, which can be empty or multiple.

ConditionalStatement: Represents if, elif, and else conditions with corresponding blocks.

LoopStatement: Represents for and while loops.

InputStatement: Captures input using the CIN keyword.

OutputStatement: Outputs data using the SHOW keyword.

ExpressionStatement: A statement that is an expression.

ReturnStatement: A return statement returning an expression.

Expression: Defines arithmetic operations, parentheses for grouping, numbers (integer and float), and variables.

Learning Outcomes:

(1) Gained a comprehensive understanding of the roles of Flex and Bison in compiler construction.

(2) Learned how to define and implement lexical rules for tokenizing custom language syntax.

(3) Developed skills in writing context-free grammar for parsing structured input.

(4) Acquired hands-on experience in generating and integrating a lexer and parser.

(5) Understood the process of handling tokens and semantic actions for syntax validation.

(6) Enhanced problem-solving skills by debugging and refining grammar and lexical definitions.

(7) Explored the design of a custom programming language, including unique syntax and semantics.

(8) Improved proficiency in C programming by integrating lexer and parser components.

(9) Learned how to manage and interpret syntax errors through Bison's error handling.

(10) Strengthened knowledge of compiler design principles and the interplay between lexical analysis and parsing.

Discussion:

Throughout this project, I had immersed myself in the intricate process of designing a custom programming language using Flex and Bison. I had started by defining the lexical rules, which enabled me to tokenize the language's syntax effectively. The challenges I faced in this phase included handling various data types and operators, which I had meticulously resolved through refining the lexer definitions. As I progressed, I had written the context-free grammar, capturing the language’s structure and rules. The integration of the lexer and parser was particularly rewarding, as it marked a significant milestone where the language began to take shape. Debugging and testing had been crucial steps, allowing me

to ensure that the language behaved as expected. By the end of the project, I had gained a deeper appreciation for compiler construction and the complexities involved in language design.

Conclusion:

This project had provided me with a thorough understanding of the fundamental concepts of compiler design, particularly the roles of lexical analysis and parsing. By utilizing Flex and Bison, I had successfully developed a lexer and parser for my custom programming language, which demonstrated a working model of how modern programming languages operate. The experience of tackling real-world problems, such as syntax error handling and token generation, had enriched my learning journey. This project had also underscored the importance of attention to detail and iterative refinement in software development. Overall, I had not only gained practical skills in using compiler tools but also developed a structured approach to solving complex programming challenges, which would undoubtedly benefit me in future projects and endeavors.

References:

1) https://www.cse.scu.edu/~m1wang/compiler/TutorialFlexBison.pdf/
2) https://begriffs.com/posts/2021-11-28-practical-parsing.html/
3) https://web.iitd.ac.in/~sumeet/flex__bison.pdf/
4) https://aquamentus.com/flex_bison.html/
5) https://www.cs.virginia.edu/~cr4bd/flex-manual/
6) https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/

----------0----------