
KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Engineering

(CSE)

# Report on
# "Project of Compiler Design Using Flex And Bison"

**Course Name:** Compiler Design Laboratory
**Course No:** CSE-3212

## Submitted To

**Nazia Jahan Khan Chowdhury**
Assistant Professor,
Department of Computer Science and Engineering, KUET

**Dipannita Biswas**
Lecturer,
Department of Computer Science and Engineering, KUET

## Submitted By

**Atiqul Islam Atik**
**Roll:** 1907107
**Year:** 3rd year, 2nd semester
Department of Computer Science and Engineering, KUET

**Introduction:**

A compiler is a crucial tool in computer science that converts high-level programming languages into machine code or lower-level languages, enabling computer execution. It involves lexical, syntax, semantic, code generation, and optimization phases. Compilers play a vital role in software development, converting human-readable code into hardware-readable formats, ensuring efficient program execution and facilitating complex algorithm translation.

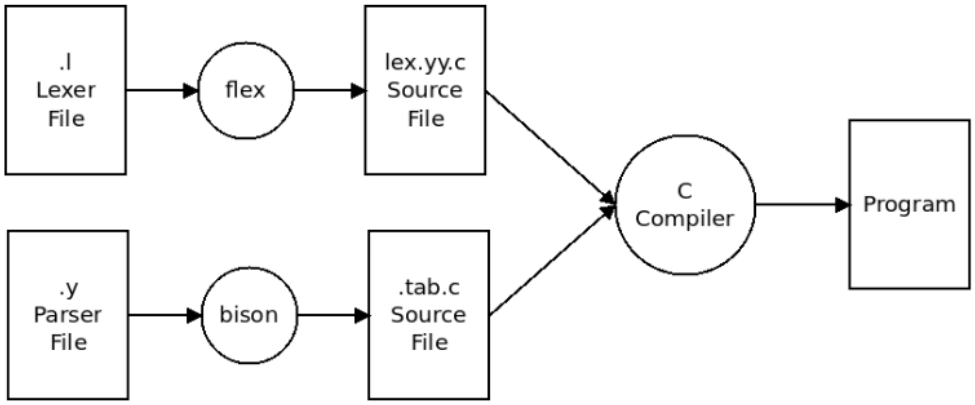
**Flex:**

Flex is a powerful tool used in compiler construction and text processing. It generates programs in C or C++ that recognize text patterns using regular expressions. Flex is commonly used in the initial stages of compiler design for lexical analysis, helping create scanners that break down source code into meaningful units like keywords, identifiers, operators, and literals. Flex's flexibility, efficiency, and ease of use make it a preferred tool for implementing lexical analyzers in various software applications.

**Bison:**

Bison is a compiler tool that generates parsers for context-free grammars. It takes a formal grammar specification as input and generates C or C++ code for a parser that can analyze the syntax based on the provided grammar rules. Bison is crucial in compiler design, particularly in the parsing phase, as it helps create parsers that understand programming language structure and syntax. It simplifies the development of compilers and interpreters by automating the creation of parsers from formal grammars, saving time and effort in handling complex syntactic structures.

**Compilation process:**




**Running Bison and Flex Program:**

1) bison -d filename.y
2) flex filename.l
3) gcc filename.tab.c lex.yy.c -o output_executable
4) ./output_executable

**Project Details:**

Input String: **Main** Token: **MAIN** Description: Indicates the start of the main function.

Input String: **{** Token: **BST** Description: Token for the opening curly brace.

Input String: **INT** Token: **INT** Description: Token indicating the declaration of an integer variable.

Input String: **a** Token: **VAR** Description: Represents a variable name.

Input String: **,** Token: **,** Description: Represents a comma used for separating variables in declarations.

Input String: **b** Token: **VAR** Description: Represents another variable name.

Input String: , Token: , Description: Represents a comma used for separating variables in declarations.

Input String: c Token: VAR Description: Represents another variable name.

Input String: ; Token: SM Description: Indicates the end of a statement.

Input String: FLOAT Token: FLOAT Description: Token indicating the declaration of a floating-point variable.

Input String: x Token: VAR Description: Represents a variable name.

Input String: , Token: , Description: Represents a comma used for separating variables in declarations.

Input String: z Token: VAR Description: Represents another variable name.

Input String: ; Token: SM Description: Indicates the end of a statement.

Input String: CHAR Token: CHAR Description: Token indicating the declaration of a character variable.

Input String: p Token: VAR Description: Represents a variable name.

Input String: ; Token: SM Description: Indicates the end of a statement.

Input String: CHAR Token: CHAR Description: Token indicating the declaration of a character variable.

Input String: q Token: VAR Description: Represents a variable name.

Input String: , Token: , Description: Represents a comma used for separating variables in declarations.

Input String: r Token: VAR Description: Represents another variable name.

Input String: ; Token: SM Description: Indicates the end of a statement.

Input String: Display Token: PRINTFUNCTION Description: Indicates a display/print function.

Input String: ( Token: PST Description: Token for the opening parenthesis.

Input String: "variable declared properly" Token: STRING_LITERAL Description: Represents a string literal.

Input String: ) Token: PEN Description: Token for the closing parenthesis.

Input String: ; Token: SM Description: Indicates the end of a statement.

Input String: a Token: VAR Description: Represents a variable name.

Input String: = Token: = Description: Represents the assignment operator.

Input String: 15 Token: NUM Description: Represents a numerical value.

Input String: ; Token: SM Description: Indicates the end of a statement.

Input String: b Token: VAR Description: Represents a variable name.

Input String: = Token: = Description: Represents the assignment operator.

Input String: 7 Token: NUM Description: Represents a numerical value.

Input String: ; Token: SM Description: Indicates the end of a statement.

Input String: c Token: VAR Description: Represents a variable name.

Input String: = Token: = Description: Represents the assignment operator.

Input String: 100 Token: NUM Description: Represents a numerical value.

Input String: ; Token: SM Description: Indicates the end of a statement.

Input String: if Token: IF Description: Indicates the start of an if statement.

Input String: ( Token: PST Description: Token for the opening parenthesis.

Input String: a Token: VAR Description: Represents a variable name.

Input String: > Token: GT Description: Represents greater than comparison.

Input String: b Token: VAR Description: Represents another variable name.

Input String: ) Token: PEN Description: Token for the closing parenthesis.

Input String: { Token: BST Description: Token for the opening curly brace.

Input String: Display Token: PRINTFUNCTION Description: Indicates a display/print function.

Input String: ( Token: PST Description: Token for the opening parenthesis.

Input String: "a is greater than b" Token: STRING_LITERAL Description: Represents a string literal.

Input String: ) Token: PEN Description: Token for the closing parenthesis.

Input String: ; Token: SM Description: Indicates the end of a statement.

Input String: } Token: BEN Description: Token for the closing curly brace.

Input String: else Token: ELSE Description: Indicates the start of an else statement.

Input String: **{** Token: **BST** Description: Token for the opening curly brace.

Input String: **Display** Token: **PRINTFUNCTION** Description: Indicates a display/print function.

Input String: **(** Token: **PST** Description: Token for the opening parenthesis.

Input String: **"a is less than b"** Token: **STRING_LITERAL** Description: Represents a string literal.

Input String: **)** Token: **PEN** Description: Token for the closing parenthesis.

Input String: **;** Token: **SM** Description: Indicates the end of a statement.

Input String: **}** Token: **BEN** Description: Token for the closing curly brace.

Input String: **elseif** Token: **ELSEIF** Description: Indicates the start of an else-if statement.

Input String: **Display** Token: **PRINTFUNCTION** Description: Indicates a display/print function.

Input String: **(** Token: **PST** Description: Token for the opening parenthesis.

Input String: **"hrllo"** Token: **STRING_LITERAL** Description: Represents a string literal.

Input String: **)** Token: **PEN** Description: Token for the closing parenthesis.

Input String: **;** Token: **SM** Description: Indicates the end of a statement.

Input String: **if** Token: **IF** Description: Indicates the start of an if statement.

Input String: **(** Token: **PST** Description: Token for the opening parenthesis.

Input String: **a** Token: **VAR** Description: Represents a variable name.

Input String: **<** Token: **LT** Description: Represents less than comparison.

Input String: **b** Token: **VAR** Description: Represents another variable name.

Input String: **)** Token: **PEN** Description: Token for the closing parenthesis.

Input String: **Display** Token: **PRINTFUNCTION** Description: Indicates a display/print function.

Input String: **(** Token: **PST** Description: Token for the opening parenthesis.

Input String: **"a is less than b"** Token: **STRING_LITERAL** Description: Represents a string literal.

Input String: **)** Token: **PEN** Description: Token for the closing parenthesis.

Input String: **;** Token: **SM** Description: Indicates the end of a statement.

Input String: **elseif** Token: **ELSEIF** Description: Indicates the start of an else-if statement.

Input String: **(** Token: **PST** Description: Token for the opening parenthesis.

Input String: **a** Token: **VAR** Description: Represents a variable name.

Input String: **>** Token: **GT** Description: Represents greater than comparison.

Input String: **c** Token: **VAR** Description: Represents another variable name.

Input String: **)** Token: **PEN** Description: Token for the closing parenthesis.

Input String: **Display** Token: **PRINTFUNCTION** Description: Indicates a display/print function.

Input String: **(** Token: **PST** Description: Token for the opening parenthesis.

Input String: **"a is greater than b"** Token: **STRING_LITERAL** Description: Represents a string literal.

Input String: **)** Token: **PEN** Description: Token for the closing parenthesis.

Input String: **;** Token: **SM** Description: Indicates the end of a statement.

**Feature of this project:**

1. Header File
2. Variable declaration
3. Variable Assignment
4. If else nested block
5. While loop, For loop
6. Switch case
7. Try Catch
8. Print function
9. Single and Multiline comment
10. Function
11. Build in function (Prime, Factorial, odd, even)

## Discussion and Conclusion:

In conclusion, while the provided code attempts to outline basic programming language constructs, it falls short in completeness, correctness, and functionality. It seems more like an incomplete demonstration or an early-stage prototype rather than a functional programming project. For a comprehensive programming language or interpreter, a more detailed and accurate implementation of syntax, semantics, data structures, error handling, and functionality is necessary