CSM 3202: Compiler Lab

Lab Report 5: The construction of Three Address Code for context-free grammars.

Name: MD. TAMIM AHMED FAHIM

ID: 2209024

Level: 3, Semester: 2

Bioinformatics Engineering

Bangladesh Agricultural University

Report submitted: February 3 ,2026

This is a documentation to the Lab 5 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 of Three Address Code for context-free grammars.

2 Tasks and Implementations

2.1 Task 1:Three-Address Code Generation for Arithmetic and Logical Statements

2.1.1 Objective

The objective of this task is to design and implement a program that reads one or more statements from an input file and generates three-address intermediate code (ICG) for each statement. The program should correctly handle arithmetic, assignment, and logical operations while respecting operator precedence and associativity. This task demonstrates the use of Lex (Flex) and Yacc (Bison) in compiler design for syntax analysis and intermediate code generation.

2.1.2 Grammar

Grammar Rules:
Program → StatementList
StatementList → Statement
| StatementList NEWLINE Statement

Statement → ID '=' Expression
| ID OpAssign Expression

OpAssign → '+=' | '-=' | '*=' | '/=' | '
Expression → Expression '+' Term
| Expression '-' Term
| Expression '||' Term
| Expression '' Term
| Expression '>' Term
| Expression '<' Term
| Term

Term → Term '*' Factor
| Term '/' Factor

| Term ’//’ Factor
| Term ’%’ Factor
| Factor

Factor → Factor ’**’ Unary
| Unary

Unary → ’!’ Unary
| ’-’ Unary
| Primary

Primary → ID
| NUM
| ’(’ Expression ’)’

ID → [a-zA-Z][a-zA-Z0-9]*
NUM → [0-9]+
NEWLINE → ” *The grammar used in this task defines the syntactic structure of valid statements and expressions with arithmetic and logical operators. It supports multiple statements separated by new lines, assignment operations, arithmetic expressions, logical expressions, unary operations, and compound assignment operators.*

Grammar Rules:
Program → StatementList
StatementList → Statement
| StatementList NEWLINE Statement

Statement → ID ’=’ Expression
| ID OpAssign Expression

OpAssign → ’+=’ | ’-=’ | ’=’ | ’/=’ | ’*
Expression → Expression ’+’ Term
| Expression ’-’ Term
| Expression ’||’ Term
| Expression ’’ Term
| Expression ’>’ Term
| Expression ’<’ Term
| Term

Term → Term ’’ Factor*

| Term '/' Factor
| Term '//' Factor
| Term '%' Factor
| Factor

*Factor → Factor '**' Unary*
| Unary

Unary → '!' Unary
| '-' Unary
| Primary

Primary → ID
| NUM
| '(' Expression ')'

*ID → [a-zA-Z][a-zA-Z0-9]**
NUM → [0-9]+
NEWLINE → "

2.1.3 Requirements

Software Requirements:

- *Flex (Lexical Analyzer Generator),*
- *Bison (Parser Generator),*
- *GCC Compiler,*
- *Operating System: Windows, Linux,*
- *text editor(VScode)*

Input Requirements:

- *An input.txt file containing valid statements using supported operators*

2.1.4 Installation and Set-up

- *Install Flex, Bison, and GCC,*
- *Place the files in the same directory,*
- *Build the project using:MakeFile,*
- *Run the program.*

2.1.5 Implementation with GitHub Link

The implementation is done using Flex for lexical analysis and Bison for syntax analysis and intermediate code generation. The lexer identifies tokens such as identifiers, numbers, operators, and new lines. The parser applies grammar rules and generates three-address code using temporary variables.

Click here

2.1.6 Input & Output

Input:

```
a = 5 + 3
b += a * 2
c = ! b || 0
d = a ** 2
e //= 3
f = ( a + b ) * ( c - d ) / e
g %= ( f ** 2) + 1
h = !(( a > b ) && ( c < d ) ) || e
i **= 2
j = i // (a + b * c)

```

Output:

```
1 t1 = 5 + 3
2 a = t1
3 t2 = a * 2
4 t3 = b + t2
5 b = t3
6 t4 = ! b
7 t5 = t4 || 0
8 c = t5
9 t6 = a ** 2
10 d = t6
11 t7 = e // 3
12 e = t7
13 t8 = a + b
14 t9 = c - d
15 t10 = t8 * t9
```

```
16 t11 = t10 / e
17 f = t11
18 t12 = f ** 2
19 t13 = t12 + 1
20 t14 = g % t13
21 g = t14
22 t15 = a > b
23 t16 = c < d
24 t17 = t15 && t16
25 t18 = ! t17
26 t19 = t18 || e
27 h = t19
28 t20 = i ** 2
29 i = t20
30 t21 = b * c
31 t22 = a + t21
32 t23 = i // t22
```

2.1.7 Working Principles

- *Lexical Analysis (Flex):Converts characters into tokens (ID, NUM, operators, functions)*
- *Syntax Analysis (Bison):*
  1.Applies the grammar to ensure the statements are valid.
  2.Handles operator precedence and function calls.
- *Intermediate Code Generation (TAC):*
  1.For each arithmetic or function expression, generates a temporary variable for intermediate computation.
  2.Assignments to variables use the temporary values if needed.
- *Output: Prints each TAC instruction sequentially, line by line.*

2.2 Task 2:Compiler Frontend for Math Functions

2.2.1 Objective

Reads arithmetic statements from input.txt, potentially spanning multiple lines.

*Supports basic operators (+, -, *, /, %), parentheses, and math functions: sqrt(), pow(), log(), exp(), sin(), cos(), tan(), abs().*

Generates Three-Address Code (TAC) for each statement, using temporary variables (t1, t2, ...) for intermediate expressions. The goal of Task 2 is to design a compiler frontend that:

Reads arithmetic statements from input.txt, potentially spanning multiple lines.

*Supports basic operators (+, -, *, /, %), parentheses, and math functions: sqrt(), pow(), log(), exp(), sin(), cos(), tan(), abs().*

Generates Three-Address Code (TAC) for each statement, using temporary variables (t1, t2, ...) for intermediate expressions.

2.2.2 Grammar

Statement → ID '=' Expression

Expression → Expression '+' Term | Expression '-' Term | Term

Term → Term '' Factor | Term '/' Factor | Term '%' Factor | Factor*

Factor → FunctionCall | '(' Expression ')' | ID | NUM | '-' Factor

FunctionCall → 'sqrt' '(' Expression ')' | 'pow' '(' Expression ',' Expression ')' | 'log' '(' Expression ')' | 'exp' '(' Expression ')' | 'sin' '(' Expression ')' | 'cos' '(' Expression ')' | 'tan' '(' Expression ')' | 'abs' '(' Expression ')'

*ID → [a-zA-Z][a-zA-Z0-9]**
NUM → [0-9]+
NEWLINE → " Program → StatementList
StatementList → Statement | StatementList NEWLINE Statement

Statement → ID '=' Expression

Expression → Expression '+' Term | Expression '-' Term | Term

Term → Term '' Factor | Term '/' Factor | Term '%' Factor | Factor*

Factor → FunctionCall | '(' Expression ')' | ID | NUM | '-' Factor

FunctionCall → 'sqrt' '(' Expression ')' | 'pow' '(' Expression ',' Expression ')' | 'log' '(' Expression ')' | 'exp' '(' Expression ')' | 'sin' '(' Expression ')' | 'cos' '(' Expression ')' | 'tan' '(' Expression ')' | 'abs' '(' Expression ')'

*ID → [a-zA-Z][a-zA-Z0-9]**
NUM → [0-9]+
NEWLINE → ''

2.2.3 Requirements

Software Requirements:

- *Flex (Lexical Analyzer Generator),*
- *Bison (Parser Generator),*
- *GCC Compiler,*
- *Operating System: Windows, Linux,*
- *text editor(VScode)*

Input Requirements:

- *An input.txt file containing valid statements using supported operators*

2.2.4 Installation and Set-up

- *Install Flex, Bison, and GCC,*
- *Place the files in the same directory,*
- *Build the project using:MakeFile,*
- *Run the program.*

2.2.5 Implementation with GitHub Link

The implementation is done using Flex for lexical analysis and Bison for syntax analysis and intermediate code generation. The lexer identifies tokens such as identifiers, numbers, operators, and new lines. The parser applies grammar rules and generates three-address code using temporary variables.

Click here

2.2.6 Input & Output

Input:

```
a = 9
b = sqrt ( a )
c = pow ( a , 3 )
d = log ( b ) + sin ( a )
e = cos ( c ) * tan ( d )
f = abs ( - a + b ) / exp ( 2 )

```

Output:

```
1 a = 9
2 t1 = sqrt ( a )
3 b = t1
4 t2 = pow ( a , 3 )
5 c = t2
6 t3 = log ( b )
7 t4 = sin ( a )
8 t5 = t3 + t4
9 d = t5
10 t6 = cos ( c )
11 t7 = tan ( d )
12 t8 = t6 * t7
13 e = t8
14 t9 = -a
15 t10 = t9 + b
16 t11 = abs ( t10 )
17 t12 = exp ( 2 )
18 t13 = t11 / t12
19 f = t13


```

2.2.7 Working Principles

- *Lexical Analysis (Flex):Converts characters into tokens (ID, NUM, operators, functions)*

- *Syntax Analysis (Bison):*
  1.Applies the grammar to ensure the statements are valid.
  2.Handles operator precedence and function calls.

- *Intermediate Code Generation (TAC):*
  1.For each arithmetic or function expression, generates a temporary variable for intermediate computation.
  2.Assignments to variables use the temporary values if needed.

- *Output: Prints each TAC instruction sequentially, line by line.*

2.2.8 Conclusion

In this assignment, we successfully designed and implemented compiler frontends for generating three-address code (TAC) from arithmetic, logical, and mathematical statements. The tasks demonstrated key principles of compiler design, including lexical analysis using Flex, syntax analysis using Bison, operator precedence, unary operations, and function handling. By generating TAC, we translated high-level expressions into a lower-level intermediate representation, which is essential for further compiler phases such as optimization and code generation. This exercise enhanced our understanding of grammar-based parsing, intermediate code generation, and the practical implementation of compiler components.