CSM 3202: Compiler Lab

Lab Report 6:Code Generation – Three Address Code and Assembly Generation for Context-Free Grammars

Name: MD. TAMIM AHMED FAHIM

ID: 2209024

Level: 3, Semester: 2

Bioinformatics Engineering

Bangladesh Agricultural University

Report submitted: February 8 ,2026

This is a documentation to the Lab 6 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:Generation of Three Address Code and Assembly Code for Arithmetic, Assignment, and Logical

2.1.1 Objective

The objective of this lab is to design and implement a code generation module for a context-free grammar that translates high-level program statements into Three Address Code (TAC) and corresponding Assembly code. The program reads single or multiple statements from an input file, properly handles new lines, and enforces correct operator precedence and associativity. It supports arithmetic, assignment, compound assignment, exponentiation, integer division, and logical operations as specified in the grammar. This lab helps in understanding syntax-directed translation, intermediate code representation, and the fundamentals of code generation in compiler design.

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 | Term
Term → 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 MOV R0, #5
2 MOV R0, #3
3 ADD R0, #3
4 MOV a, R0

5 MOV R0, a
6 MOV R0, #2
7 MUL R0, #2
10 MOV R1, b
9 ADD R1, R0
8 MOV b, R1

11 MOV R0, b
12 NOT R0
13 MOV R0, #0
14 OR R0, #0
15 MOV c, R0

16 MOV R0, a
17 MOV R0, #2
18 POW R0, #2
19 MOV d, R0

20 MOV R0, #3
23 MOV R1, e
22 IDIV R1, R0
21 MOV e, R1

24 MOV R0, a
25 MOV R0, b
26 ADD R0, b
27 MOV R0, c
```

2.1.7 Working Principle

- **Input Processing:** The program reads single or multiple statements line by line from `input.txt`.
- **Lexical Analysis (Flex):** Converts characters into tokens such as identifiers (ID), numbers (NUM), arithmetic, logical, and assignment operators.
- **Syntax Analysis (Bison):** Parses the tokens using the given grammar, ensuring syntactic correctness and proper operator precedence and associativity.
- **Three Address Code Generation:** Generates TAC instructions using temporary variables for intermediate results of arithmetic, logical, and assignment operations.
- **Assembly Code Generation:** Converts each TAC instruction into corresponding Assembly-level instructions using registers and basic operations.

2.2 Task 2:Generation of Three Address Code and Assembly Code for Arithmetic Expressions with Mathematical Functions

2.2.1 Objective

The objective of Task 2 is to design and implement a compiler front-end that reads arithmetic assignment statements with mathematical function calls from an input file. The program validates the syntax using the given grammar, handles operator precedence and parentheses, and supports built-in mathematical functions such as sqrt(), pow(), log(), exp(), sin(), cos(), tan(), and abs(). For each valid statement, the compiler generates corresponding Three Address Code (TAC) and Assembly code, demonstrating the translation of high-level mathematical expressions into intermediate and low-level representations.

2.2.2 Grammar

Program → StatementList
StatementList → Statement | StatementList NEWLINE Statement
Statement → ID = Expression
Expression → Expression + Term
| Expression − Term
| Term
Term → Term ∗ Factor
| Term / Factor
| Term | 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:

```
MOV R0, #9
MOV a, R0

MOV R0, a
SQRT R0
MOV b, R0

MOV R0, a
MOV R1, #3
POW R0, R1
MOV c, R0

MOV R0, b
LOG R0
MOV R1, a
SIN R1
ADD R0, R1
MOV d, R0

MOV R0, c
COS R0
MOV R1, d
TAN R1
MUL R0, R1
MOV e, R1

MOV R0, a
NEG R0
MOV R1, b
ADD R0, R1
ABS R0
MOV R1, #2
EXP R1
DIV R0, R1
MOV f, R1
```

2.2.7 Working Principle

- **Input Processing:** The program reads single or multiple statements line by line from `input.txt`.
- **Lexical Analysis (Flex):** Converts characters into tokens such as identifiers (ID), numbers (NUM), arithmetic, logical, and assignment operators.
- **Syntax Analysis (Bison):** Parses the tokens using the given grammar, ensuring syntactic correctness and proper operator precedence and associativity.
- **Three Address Code Generation:** Generates TAC instructions using temporary variables for intermediate results of arithmetic, logical, and assignment operations.
- **Assembly Code Generation:** Converts each TAC instruction into corresponding Assembly-level instructions using registers and basic operations.

2.2.8 Conclusion

In this lab, compiler front-ends were successfully implemented for generating Three Address Code (TAC) and Assembly code from high-level statements. Task 1 focused on arithmetic, assignment, and logical operations, handling multiple statements, operator precedence, and various assignment operators. Task 2 extended this to arithmetic expressions with mathematical function calls, correctly processing nested expressions, operator precedence, and function evaluations. Together, these tasks demonstrated the principles of syntax-directed translation, intermediate code generation, and low-level code mapping, providing a clear understanding of how high-level language constructs are converted into executable code in the code generation phase of a compiler.