

# Compiler Construction

CS-471L: Compiler Construction Lab



**Project Supervisor**

Mr. Laeeq Khan Niazi

**Developer:**

Amad Irfan

2021-CS-25

**University of Engineering and Technology  
Lahore, Pakistan**

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Purpose and Motivation . . . . .	3
1.2	Objectives . . . . .	3
1.3	Features . . . . .	3
1.4	Additional Features . . . . .	4
1.5	Compiler Overview . . . . .	4
1.6	Phases of Compilation . . . . .	4
1.7	Applications of Compilers . . . . .	4
<b>2</b>	<b>Implementation Details</b>	<b>5</b>
2.1	Enums and Symbol Table . . . . .	5
2.2	Three-Address Code (TAC) . . . . .	5
2.3	Lexer . . . . .	5
2.4	Parser . . . . .	5
2.5	Syntax Tree Generation . . . . .	5
<b>3</b>	<b>Testing and Results</b>	<b>6</b>
3.1	Sample Input . . . . .	6
3.2	Sample Output . . . . .	7
3.3	Syntax Tree Output . . . . .	8
<b>4</b>	<b>Conclusion</b>	<b>9</b>

# 1 Introduction

## 1.1 Purpose and Motivation

The primary objective of this project is to design a custom compiler, showcasing the complete process of translating high-level language code into low-level machine code. Compilers are pivotal tools in software development, enabling programmers to write efficient, portable, and maintainable applications. This project bridges the gap between theoretical knowledge of language processing and practical implementation.

Constructing a compiler provides insight into crucial aspects like code generation, optimization, and error handling. The motivation stems from the curiosity to understand how human-readable code is transformed into machine-executable instructions. By designing a compiler, this project not only reinforces concepts such as automata and grammar but also highlights optimization techniques and error-detection mechanisms implemented in modern compilers.

## 1.2 Objectives

The project aims to:

- Implement a compiler capable of translating high-level language into machine-readable code.
- Develop a grammar-based parser to validate syntactic structures and handle errors effectively.
- Create a module for intermediate representation (IR) generation, ensuring platform independence and enabling optimizations.
- Design a modular compiler architecture encompassing all key phases of compilation.
- Provide detailed compile-time feedback, including warnings and suggestions, to aid user understanding of the source code.

## 1.3 Features

This compiler includes the following features:

- **Tokenization:** The lexer converts the source code into tokens, which are the smallest units of meaning, such as keywords, identifiers, and operators.
- **Comment Handling:** Supports both single-line (`//`) and multi-line (`/* ... */`) comments.
- **Whitespace Management:** Ignores whitespace while tracking line numbers for precise error reporting.
- **Recognition of Literals and Identifiers:** Identifies numeric literals, strings, characters, and variable names.
- **Keyword Recognition:** Recognizes common keywords such as `int`, `float`, `if`, `else`, `while`, and `return`.
- **Syntax Error Reporting:** Generates meaningful error messages with line numbers for debugging.

- **Intermediate Representation:** Produces Three-Address Code (TAC) for platform-independent code generation.
- **Assembly Code Generation:** Translates TAC into basic assembly-like instructions.

## 1.4 Additional Features

In addition to the above, the compiler includes:

- **Floating-Point Number Support:** Handles `float` literals alongside integers.
- **String and Character Support:** Recognizes and processes string and character literals.
- **Syntax Tree Generation:** Constructs a syntax tree representing the syntactic structure of the input.

## 1.5 Compiler Overview

A compiler is a software tool that converts programs written in high-level programming languages into low-level machine code. It plays a critical role in enabling software development for diverse platforms, optimizing performance, and detecting errors. This project encapsulates the complete workflow of a compiler, from lexical analysis to assembly code generation.

## 1.6 Phases of Compilation

The compilation process consists of the following phases:

1. **Lexical Analysis:** Reads the source code and tokenizes it into meaningful units.
2. **Syntax Analysis:** Validates the grammatical structure and generates a parse tree.
3. **Semantic Analysis:** Checks the logical consistency and type correctness of the program.
4. **Intermediate Code Generation:** Produces an IR for platform-independent optimization and translation.
5. **Optimization:** Enhances the IR for improved runtime performance.
6. **Code Generation:** Translates the optimized IR into target machine code.

## 1.7 Applications of Compilers

Compilers are utilized in:

- Translating programming languages (e.g., from C++ to Java bytecode).
- Developing high-performance applications with tools like GCC and LLVM.
- Programming embedded systems with constrained resources.
- Optimizing machine learning frameworks such as TensorFlow.

## 2 Implementation Details

### 2.1 Enums and Symbol Table

Enums define token types for keywords, operators, and literals. The **SymbolTable** class manages variable information, including names, types, and scope levels. It prevents variable redefinition and provides efficient lookup using an unordered map.

### 2.2 Three-Address Code (TAC)

TAC is used for intermediate representation. Each TAC instruction includes an operator, two operands, and a result. The **AssemblyGenerator** translates TAC into assembly-like instructions with register allocation.

### 2.3 Lexer

The **Lexer** class tokenizes source code into keywords, identifiers, literals, and operators. It skips comments and whitespace while providing error handling for invalid syntax.

### 2.4 Parser

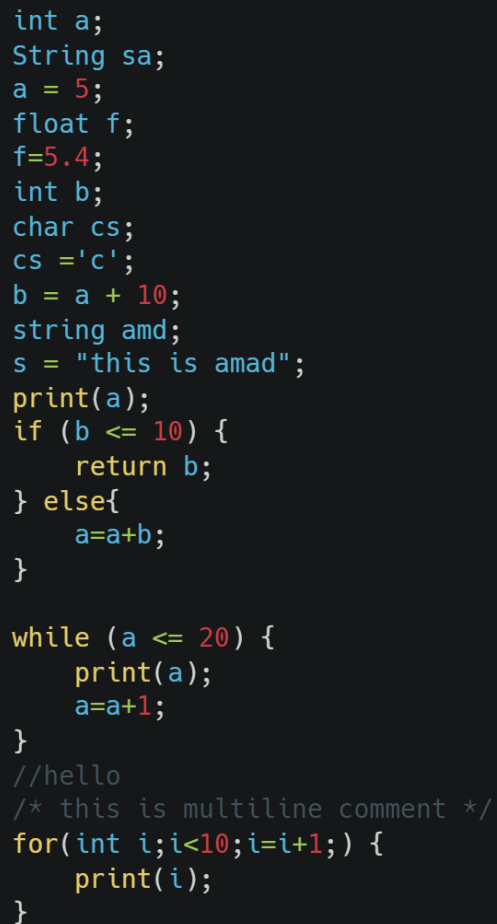
The **Parser** validates syntax, constructs TAC, and manages the symbol table. It supports declarations, assignments, control structures, and expressions.

### 2.5 Syntax Tree Generation

The syntax tree represents the hierarchical structure of the program. Each node in the tree corresponds to a construct such as a declaration or a loop.

## 3 Testing and Results

### 3.1 Sample Input



```
int a;
String sa;
a = 5;
float f;
f=5.4;
int b;
char cs;
cs ='c';
b = a + 10;
string amd;
s = "this is amad";
print(a);
if (b <= 10) {
    return b;
} else{
    a=a+b;
}

while (a <= 20) {
    print(a);
    a=a+1;
}
//hello
/* this is multiline comment */
for(int i;i<10;i=i+1;) {
    print(i);
}
```

Figure 1: Sample Input Program

## 3.2 Sample Output

```

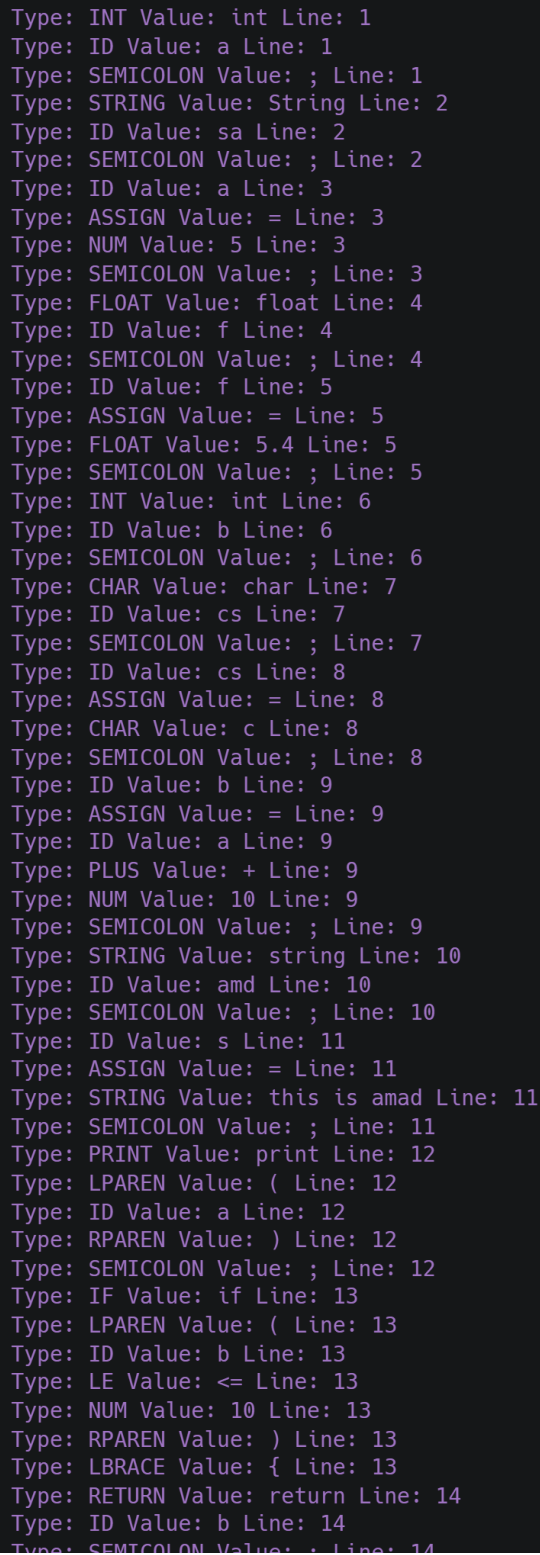
● ● ●

= 5, => a
= 5.4, => f
= c, => cs
+ a, 10 => t3
= t3, => b
= this is amad, => s
print a, =>
<= b, 10 => t7
return b, =>
+ a, b => t9
= t9, => a
<= a, 20 => t11
print a, =>
+ a, 1 => t13
= t13, => a
< i, 10 => t15
+ i, 1 => t16
= t16, => i
print i, =>
Symbol Table:
-----
Name      Type      Scope Level
-----
i          int       25
sa         String    2
f          float     4
b          int       6
amd        string    10
a          int       1
cs         char      7
-----

Generated Assembly Code is:
MOV R0, 5
MOV R1, 5.4
MOV R2, c
ADD R0,
MOV R3, R0
MOV R4, t3
MOV R5, this is amad
MOV EAX, R4
RET
ADD R0, R4
MOV R8, R0
MOV R0, t9
ADD R0,
MOV R10, R0
MOV R0, t13
ADD ,
MOV R12,
MOV , t16
```

Figure 2: Output Tokens and Generated Assembly

### 3.3 Syntax Tree Output



```
Type: INT Value: int Line: 1
Type: ID Value: a Line: 1
Type: SEMICOLON Value: ; Line: 1
Type: STRING Value: String Line: 2
Type: ID Value: sa Line: 2
Type: SEMICOLON Value: ; Line: 2
Type: ID Value: a Line: 3
Type: ASSIGN Value: = Line: 3
Type: NUM Value: 5 Line: 3
Type: SEMICOLON Value: ; Line: 3
Type: FLOAT Value: float Line: 4
Type: ID Value: f Line: 4
Type: SEMICOLON Value: ; Line: 4
Type: ID Value: f Line: 5
Type: ASSIGN Value: = Line: 5
Type: FLOAT Value: 5.4 Line: 5
Type: SEMICOLON Value: ; Line: 5
Type: INT Value: int Line: 6
Type: ID Value: b Line: 6
Type: SEMICOLON Value: ; Line: 6
Type: CHAR Value: char Line: 7
Type: ID Value: cs Line: 7
Type: SEMICOLON Value: ; Line: 7
Type: ID Value: cs Line: 8
Type: ASSIGN Value: = Line: 8
Type: CHAR Value: c Line: 8
Type: SEMICOLON Value: ; Line: 8
Type: ID Value: b Line: 9
Type: ASSIGN Value: = Line: 9
Type: ID Value: a Line: 9
Type: PLUS Value: + Line: 9
Type: NUM Value: 10 Line: 9
Type: SEMICOLON Value: ; Line: 9
Type: STRING Value: string Line: 10
Type: ID Value: amd Line: 10
Type: SEMICOLON Value: ; Line: 10
Type: ID Value: s Line: 11
Type: ASSIGN Value: = Line: 11
Type: STRING Value: this is amad Line: 11
Type: SEMICOLON Value: ; Line: 11
Type: PRINT Value: print Line: 12
Type: LPAREN Value: ( Line: 12
Type: ID Value: a Line: 12
Type: RPAREN Value: ) Line: 12
Type: SEMICOLON Value: ; Line: 12
Type: IF Value: if Line: 13
Type: LPAREN Value: ( Line: 13
Type: ID Value: b Line: 13
Type: LE Value: <= Line: 13
Type: NUM Value: 10 Line: 13
Type: RPAREN Value: ) Line: 13
Type: LBRACE Value: { Line: 13
Type: RETURN Value: return Line: 14
Type: ID Value: b Line: 14
Type: SEMICOLON Value: ; Line: 14
```



Figure 3: Generated Syntax Tree

## 4 Conclusion

This project successfully implements a basic compiler framework. It provides functionality for lexical analysis, syntax validation, intermediate representation, and code generation. The modular architecture facilitates future enhancements, including advanced optimizations and support for additional language features.