**Final Project**

CS-465L Compiler Construction

****

**Project Supervisor**

Sir Laeeq Khan Niazi

**Project Members**

|  |  |
| --- | --- |
| M Abdullah | 2021-CS-104 |

**Computer Science Department**

**University of Engineering and Technology, Lahore**

Table of Contents

[Abstract 4](#_Toc184774687)

[Introduction 4](#_Toc184774688)

[Objectives 4](#_Toc184774689)

[Token Types: 4](#_Toc184774690)

[1. Arithmetic Operators (5) 5](#_Toc184774691)

[2. Assignment and Comparison Operators (7) 5](#_Toc184774692)

[3. Logical Operators (4) 5](#_Toc184774693)

[4. Control Structures (10) 5](#_Toc184774694)

[5. Data Types (6) 6](#_Toc184774695)

[6. Literals (4) 6](#_Toc184774696)

[7. Punctuation and Delimiters (6) 6](#_Toc184774697)

[8. Object-Oriented Keywords (4) 7](#_Toc184774698)

[9. Exception Handling (3) 7](#_Toc184774699)

[10. Pointer and Member Operators (4) 7](#_Toc184774700)

[11. End-of-File Indicator (1) 7](#_Toc184774701)

[System Design 7](#_Toc184774702)

[1. Lexer 7](#_Toc184774703)

[2. Symbol Table 8](#_Toc184774704)

[3. Intermediate Code Generator 8](#_Toc184774705)

[4. Assembly Code Generator 8](#_Toc184774706)

[Implementation 9](#_Toc184774707)

[1. Lexer Implementation 9](#_Toc184774708)

[2. Symbol Table 9](#_Toc184774709)

[3. Intermediate Code Generation 10](#_Toc184774710)

[4. Assembly Code Generation 10](#_Toc184774711)

[Results 10](#_Toc184774712)

[Input Source Code 10](#_Toc184774713)

[Generated Tokens 10](#_Toc184774714)

[Symbol Table 10](#_Toc184774715)

[Intermediate Code 10](#_Toc184774716)

[Assembly Code 11](#_Toc184774717)

[Features 11](#_Toc184774718)

[Test Cases for Compiler Implementation 11](#_Toc184774719)

[Test Case 1: Arithmetic Operators 11](#_Toc184774720)

[Test Case 2: Assignment and Comparison Operators 12](#_Toc184774721)

[Test Case 3: Logical Operators 13](#_Toc184774722)

[Test Case 4: Control Structures 14](#_Toc184774723)

[Test Case 5: Data Types 15](#_Toc184774724)

[Test Case 6: Literals 15](#_Toc184774725)

[Test Case 7: Punctuation and Delimiters 16](#_Toc184774726)

[Test Case 8: Object-Oriented Keywords 16](#_Toc184774727)

[Test Case 9: Exception Handling 16](#_Toc184774728)

[Test Case 10: Pointer and Member Operators 17](#_Toc184774729)

[Test Case 11: End-of-File Indicator 17](#_Toc184774730)

[Test Case 12: Complex Arithmetic and Logical Expression Evaluation 17](#_Toc184774731)

[GitHub: 19](#_Toc184774732)

[Conclusion 20](#_Toc184774733)

# Abstract

This project aims to develop a compiler capable of performing lexical analysis, generating intermediate code, and translating it into assembly code. The implemented compiler supports fundamental programming constructs, including arithmetic operations, control structures, and pointer manipulation. It features a symbol table for identifier tracking, a lexer for token generation, and modules for intermediate and assembly code generation. Additionally, the compiler is designed to handle multiple token types and error detection efficiently, providing a foundational understanding of compiler design principles.

# Introduction

Compilers are integral components of modern software development, translating high-level programming languages into machine-readable code. This project presents a simplified compiler that incorporates essential phases of compilation: lexical analysis, intermediate code generation, and assembly code translation. The purpose of this project is to understand and implement the core concepts of compiler construction while maintaining compatibility with basic programming constructs. The system also emphasizes modularity and extendibility for future improvements.

# Objectives

* To design and implement a lexer for tokenizing source code.
* To build a symbol table for managing variables, types, and scope information.
* To develop modules for generating intermediate code from tokens.
* To translate intermediate code into assembly instructions for execution.
* To ensure the compiler supports key programming constructs, such as arithmetic operations, control structures, and pointers.
* To incorporate error detection for unknown tokens and unsupported constructs.
* To support an extensive range of token types.

# Token Types:

This project focuses on the essential components of a compiler, dealing with lexical analysis, tokenization, symbol table management, three-address code generation, and assembly language translation. The **53** token types listed are foundational for creating a robust compiler for a programming language. Here is a breakdown of how these token types integrate into the major components of your project:

## 1. Arithmetic Operators (5)

* **PLUS** (+): Represents addition.
* **MINUS** (-): Represents subtraction.
* **MULTIPLY** (\*): Represents multiplication.
* **DIVIDE** (/): Represents division.
* **MODULO** (%): Represents the remainder operation.

## 2. Assignment and Comparison Operators (7)

* **ASSIGN** (=): Assigns a value to a variable.
* **EQUAL** (==): Compares two values for equality.
* **NOT\_EQUAL** (!=): Checks if two values are not equal.
* **LESS\_THAN** (<): Checks if the left operand is smaller than the right operand.
* **GREATER\_THAN** (>): Checks if the left operand is larger than the right operand.
* **LESS\_EQUAL** (<=): Checks if the left operand is smaller or equal to the right operand.
* **GREATER\_EQUAL** (>=): Checks if the left operand is larger or equal to the right operand.

## 3. Logical Operators (4)

* **LOGICAL\_AND** (&&): Represents logical conjunction (AND).
* **LOGICAL\_OR** (||): Represents logical disjunction (OR).
* **LOGICAL\_NOT** (!): Represents logical negation (NOT).
* **OR** (|) and **AND** (&): Bitwise OR and AND operations.

## 4. Control Structures (10)

* **IF**: Used for conditional branching.
* **ELSE**: Specifies the alternative branch of an if statement.
* **WHILE**: Represents a loop that executes while a condition is true.
* **FOR**: Represents a loop with initialization, condition, and increment.
* **RETURN**: Exits from a function, optionally returning a value.
* **BREAK**: Exits the nearest enclosing loop or switch statement.
* **CONTINUE**: Skips the current iteration of the nearest enclosing loop.
* **SWITCH**: Specifies a multi-way branch.
* **CASE**: Specifies individual conditions in a switch statement.
* **DEFAULT**: Specifies the fallback branch in a switch statement.

## 5. Data Types (6)

* **INT**: Represents integer data type.
* **FLOAT**: Represents single-precision floating-point numbers.
* **DOUBLE**: Represents double-precision floating-point numbers.
* **CHAR**: Represents character data type.
* **STRING**: Represents string data type.
* **VOID**: Specifies no value or type (used in function definitions).

## 6. Literals (4)

* **IDENTIFIER**: Represents variable names, function names, or other user-defined symbols.
* **INTEGER\_LITERAL**: Represents whole numbers.
* **FLOAT\_LITERAL**: Represents floating-point numbers.
* **STRING\_LITERAL**: Represents text enclosed in quotes.

## 7. Punctuation and Delimiters (6)

* **SEMICOLON** (;): Marks the end of a statement.
* **COMMA** (,): Separates values, arguments, or elements in a list.
* **LEFT\_PAREN** ((): Denotes the start of a grouping or function arguments.
* **RIGHT\_PAREN** ()): Denotes the end of a grouping or function arguments.
* **LEFT\_BRACE** ({): Denotes the start of a block.
* **RIGHT\_BRACE** (}): Denotes the end of a block.

## 8. Object-Oriented Keywords (4)

* **PUBLIC**: Specifies public visibility in object-oriented programming.
* **PRIVATE**: Specifies private visibility in object-oriented programming.
* **PROTECTED**: Specifies protected visibility in object-oriented programming.
* **STRUCT**: Declares a structure.

## 9. Exception Handling (3)

* **TRY**: Begins a block of code that might throw exceptions.
* **CATCH**: Handles exceptions thrown by the try block.
* **THROW**: Used to throw an exception.

## 10. Pointer and Member Operators (4)

* **REFERENCE** (&): Retrieves the address of a variable.
* **DEREFERENCE** (\*): Accesses the value at an address.
* **MEMBER\_ACCESS** (.): Accesses a member of a class or structure.
* **UNKNOWN**: Represents an unidentified or unrecognized token.

## 11. End-of-File Indicator (1)

* **END\_OF\_FILE**: Represents the termination of the input file during parsing.

# System Design

## 1. Lexer

The lexer scans the source code character by character to identify tokens. It recognizes a broad range of tokens to support fundamental programming constructs. Key features include:

* **Token Types**: Supports arithmetic operators (+, -, \*, /), relational operators (<, >, ==, !=), logical operators (&&, ||, !), assignment operators (=), delimiters (e.g., ;, {, }), and keywords (e.g., if, else, while, int, float).
* **Error Handling**: Flags unknown or invalid tokens, ensuring robust lexical analysis.
* **Line and Column Tracking**: Associates tokens with their positions in the source code for better debugging.
* **Functionality**: The lexer handles approximately 20 distinct token types, including identifiers, literals (integer, float, string), and reserved keywords.

## 2. Symbol Table

The symbol table is a critical component for managing variables, types, and scope information. It stores:

* **Identifier Name**: Name of the variable or function.
* **Type**: Data type (e.g., int, float, string).
* **Scope**: Variable visibility (e.g., global, local).
* **Line Number**: Declaration or usage location.
* **Additional Features**: Handles duplicate declarations and provides detailed error messages for scope violations.

## 3. Intermediate Code Generator

Intermediate code bridges the gap between high-level code and machine code. This project uses a three-address code (TAC) format for simplicity:

* **Operations**: Handles arithmetic (+, -, \*, /), logical (&&, ||, !), relational (<, >, ==) operations, and assignments.
* **Example**: c = a + b is translated into temp1 = a + b; c = temp1.
* **Control Flow**: Includes labels and jump instructions for if, else, while, and switch constructs.
* **Extendability**: Can be adapted to support arrays and function calls in future versions.

## 4. Assembly Code Generator

The assembly code generator translates TAC into Intel x86-64 assembly:

* **Data Movement**: mov instructions for variable assignments.
* **Arithmetic Operations**: add, sub, mul, and div instructions.
* **Control Flow**: jmp, je, and jne instructions for conditional and unconditional branching.
* **Example**:

mov rax, a

add rax, b

mov c, rax

# Implementation

## 1. Lexer Implementation

The lexer is implemented in C++ using a Lexer class. The tokenize() method processes the source code, returning a list of tokens. Error handling ensures invalid characters are flagged as UNKNOWN tokens. Regular expressions are used to identify token patterns efficiently.

## 2. Symbol Table

The symbol table is managed using a SymbolTable class. It supports:

* **Insertion**: Adds new identifiers with attributes.
* **Printing**: Displays the symbol table for debugging purposes.
* **Scope Management**: Differentiates between global and local variables.

## 3. Intermediate Code Generation

A ThreeAddressCode struct is used to represent TAC. Arithmetic and logical operations are processed by iterating over tokens and identifying patterns. The code generation module ensures modularity for future enhancements.

## 4. Assembly Code Generation

Assembly instructions are generated using a series of generateInstruction() calls. Each TAC operation is mapped to one or more assembly instructions. The generator handles both data movement and control flow efficiently.

# Results

## Input Source Code

int main() {

int a = 10;

float b = 5.5f;

int sum = a + 5;

if (a == 10) {

a = a + 1;}

## Generated Tokens

Token{type: INT, value: 'int', line: 1, column: 1}

Token{type: IDENTIFIER, value: 'main', line: 1, column: 5}

## Symbol Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier** | **Type** | **Scope** | **Line** |
| a | int | global | 2 |
| b | float | global | 3 |

## Intermediate Code

temp0 = a + 5

sum = temp0

if a == 10 goto label1

...

## Assembly Code

.intel\_syntax noprefix

.global main

main:

mov rax, 10

mov [a], rax

add rax, 5

mov [sum], rax

cmp rax, 10

je label1

# Features

1. **Extensive Token Support**: Handles approximately 53 token types, including identifiers, literals, operators, and keywords.
2. **Error Detection**: Flags unknown tokens and provides detailed error messages.
3. **Symbol Table Management**: Tracks variables with type, scope, and usage details.
4. **Intermediate Representation**: Uses three-address code for modular code generation.
5. **Assembly Translation**: Efficiently maps intermediate code to x86-64 assembly instructions.

# Test Cases for Compiler Implementation

## Test Case 1: Arithmetic Operators

**Objective:** Test the recognition and handling of all arithmetic operators.

**Input Source Code:**

int a = 10;

int b = 5;

int sum = a + b;

int diff = a - b;

int prod = a \* b;

int quotient = a / b;

int remainder = a % b;

**Output:**

* Tokens: INT, IDENTIFIER, ASSIGN, INTEGER\_LITERAL, PLUS, MINUS, MULTIPLY, DIVIDE, MODULO
* Symbol Table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier** | **Type** | **Scope** | **Line** |
| a | int | global | 1 |
| b | int | global | 2 |
| sum | int | global | 3 |
| diff | int | global | 4 |
| prod | int | global | 5 |
| quotient | int | global | 6 |
| remainder | int | global | 7 |

* Intermediate Code:

sum = a + b

diff = a - b

prod = a \* b

quotient = a / b

remainder = a % b

## Test Case 2: Assignment and Comparison Operators

**Objective:** Verify handling of assignment and comparison operators.

**Input Source Code:**

int a = 10;

int b = 20;

int isEqual = (a == b);

int isNotEqual = (a != b);

b = a;

int isLess = (a < b);

int isGreater = (a > b);

int isLessOrEqual = (a <= b);

int isGreaterOrEqual = (a >= b);

**Output:**

* Tokens: ASSIGN, EQUAL, NOT\_EQUAL, LESS\_THAN, GREATER\_THAN, LESS\_EQUAL, GREATER\_EQUAL
* Intermediate Code:

isEqual = a == b

isNotEqual = a != b

b = a

isLess = a < b

isGreater = a > b

isLessOrEqual = a <= b

isGreaterOrEqual = a >= b

## Test Case 3: Logical Operators

**Objective:** Test handling of logical and bitwise operators.

**Input Source Code:**

int a = 1;

int b = 0;

int c = (a && b);

int d = (a || b);

int e = (!a);

int f = (a | b);

int g = (a & b);

**Output:**

* Tokens: LOGICAL\_AND, LOGICAL\_OR, LOGICAL\_NOT, OR, AND
* Intermediate Code:

c = a && b

d = a || b

e = !a

f = a | b

g = a & b

## Test Case 4: Control Structures

**Objective:** Verify functionality for control structures.

**Input Source Code:**

int a = 0;

if (a == 0) {

a = 1;

} else {

a = 2;

}

while (a < 10) {

a = a + 1;

}

for (int i = 0; i < 5; i++) {

a = a \* 2;

}

return a;

**Output:**

* Tokens: IF, ELSE, WHILE, FOR, RETURN
* Intermediate Code:

if a == 0 goto label1

a = 1

goto label2

label1: a = 2

label2: while a < 10 goto label3

a = a + 1

label3: for i = 0; i < 5; i++ {

a = a \* 2

}

return a

## Test Case 5: Data Types

**Objective:** Test the recognition of various data types.

**Input Source Code:**

int a = 10;

float b = 5.5;

double c = 3.14159;

char d = 'x';

string e = "hello";

void func();

**Output:**

* Tokens: INT, FLOAT, DOUBLE, CHAR, STRING, VOID
* Symbol Table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier** | **Type** | **Scope** | **Line** |
| a | int | global | 1 |
| b | float | global | 2 |
| c | double | global | 3 |
| d | char | global | 4 |
| e | string | global | 5 |

## Test Case 6: Literals

**Objective:** Ensure correct identification of literals.

**Input Source Code:**

int a = 100;

float b = 20.5;

string c = "hello";

char d = 'x';

**Output:**

* Tokens: INTEGER\_LITERAL, FLOAT\_LITERAL, STRING\_LITERAL, CHAR

## Test Case 7: Punctuation and Delimiters

**Objective:** Test handling of delimiters and punctuation.

**Input Source Code:**

int a = (10 + 5);

if (a > 5) {

a = a + 1;

}

**Output:**

* Tokens: SEMICOLON, COMMA, LEFT\_PAREN, RIGHT\_PAREN, LEFT\_BRACE, RIGHT\_BRACE

## Test Case 8: Object-Oriented Keywords

**Objective:** Test handling of object-oriented constructs.

**Input Source Code:**

class MyClass {

public:

int x;

private:

int y;

protected:

int z;

};

**Output:**

* Tokens: PUBLIC, PRIVATE, PROTECTED, CLASS

## Test Case 9: Exception Handling

**Objective:** Verify recognition of exception handling keywords.

**Input Source Code:**

try {

throw "Error";

} catch (const char\* msg) {

return;

}

**Output:**

* Tokens: TRY, CATCH, THROW

## Test Case 10: Pointer and Member Operators

**Objective:** Ensure correct handling of pointer and member operators.

**Input Source Code:**

int\* ptr = &a;

\*ptr = 20;

obj.member;

**Output:**

* Tokens: REFERENCE, DEREFERENCE, MEMBER\_ACCESS

## Test Case 11: End-of-File Indicator

**Objective:** Test EOF recognition.

**Input Source Code:**

int a = 10;

**Output:**

* Tokens: END\_OF\_FILE
* Output: Compiler terminates after processing the last line.

## Test Case 12: Complex Arithmetic and Logical Expression Evaluation

**Objective:** Evaluate a complex mix of arithmetic and logical operations.

**Input Source Code:**

int a = 10;

int b = 5;

int c = 20;

int result = (a + b) \* c / (b - 3) && (a != b) || (c > a && b <= c);

**Expected Output:**

* Tokens: PLUS, MULTIPLY, DIVIDE, LOGICAL\_AND, LOGICAL\_OR, NOT\_EQUAL, GREATER\_THAN, LESS\_EQUAL
* Symbol Table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier** | **Type** | **Scope** | **Line** |
| a | int | global | 1 |
| b | int | global | 2 |
| c | int | global | 3 |
| result | int | global | 4 |

* Intermediate Code:

temp1 = a + b

temp2 = temp1 \* c

temp3 = b - 3

temp4 = temp2 / temp3

temp5 = a != b

temp6 = c > a

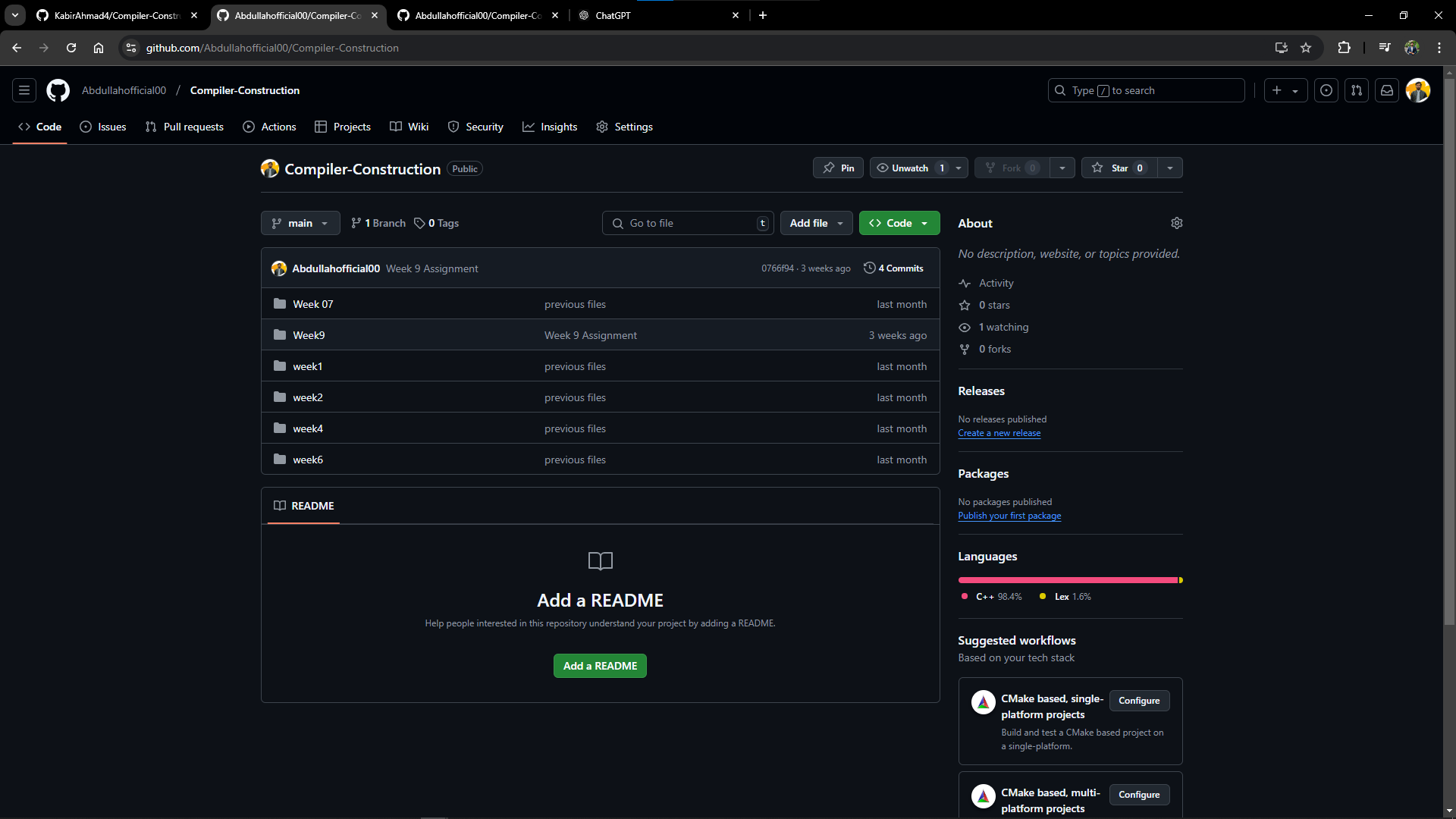
temp7 = b <= c

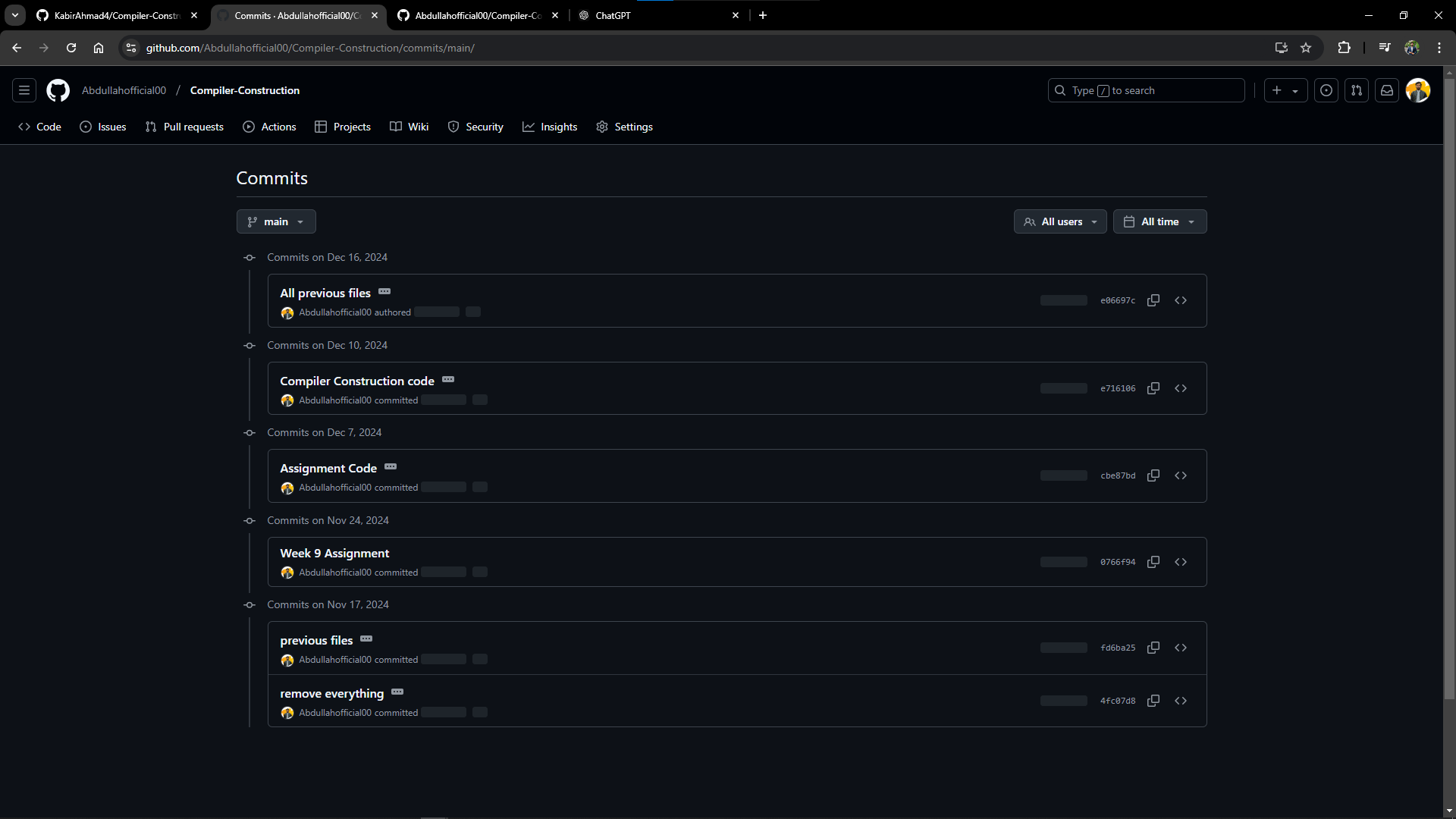
temp8 = temp6 && temp7

result = temp4 && temp5 || temp8

* Assembly Code:
* .intel\_syntax noprefix
* .global main
* main:
* push rbp
* mov rbp, rsp
* # Assignment
* mov rax, 10
* mov [a], rax
* # Assignment
* mov rax, 5
* mov [b], rax
* # Assignment
* mov rax, 20
* mov [c], rax
* # Assignment
* mov rax, (
* mov [result], rax
* # Addition
* mov rax, a
* add rax, b
* mov [temp4], rax
* # Dereferencing pointer
* mov rax, [)]
* mov [temp5], rax
* # Getting reference (address)
* lea rax, [(]
* mov [&&], rax
* # Getting reference (address)
* lea rax, [b]
* mov [&&], rax
* mov rax, 0
* leave
* ret

# GitHub:





# Conclusion

This project successfully demonstrates the design and implementation of a basic compiler. By integrating lexical analysis, symbol table management, intermediate code generation, and assembly translation, it provides a foundation for understanding compiler construction. Future improvements could include extending syntax analysis and supporting more complex constructs like function calls, arrays, and object-oriented features.