**Table of Contents**

[Table of Contents 1](#_Toc184802192)

[Compiler Development: Phase-by-Phase Breakdown 1](#_Toc184802193)

[Phase 1: Lexical Analysis (Lexer) 1](#_Toc184802194)

[Phase 2: Syntax Analysis (Parser) 3](#_Toc184802195)

[Phase 3: Semantic Analysis 5](#_Toc184802196)

[Phase 4: Intermediate Code Generation 6](#_Toc184802197)

[Phase 5: Code Generation 9](#_Toc184802198)

[Additional Features: 11](#_Toc184802199)

# Introduction:

The proposed project is to develop a compiler that can perform lexical analysis, generate intermediate code, and translate it to the assembly code. The basic programming constructs such as arithmetic operations, control structures, and pointer manipulation are supported. It has a symbol table for tracking identifiers, lexer for generating tokens, and modules for generating intermediate and assembly codes. The compiler is also designed to handle multiple token types and error detection efficiently, providing a foundational understanding of compiler design principles.

# Compiler Phases:

## Up until this point, the c++ compiler being developed cycles through several key phases in converting the source code into target machine code. These phases ensure the code is correctly processed and executed efficiently. They include the following:

## Lexical Analysis (Tokenizer)

## Syntax Analysis (Parser)

## Semantic Analysis

## Intermediate Code Generation

## Code Generation

## Each phase performs a distinct action, from breaking up the code into tokens to the generation of target code.

## 01: Lexical Analysis (Tokenizer)

**Description**: This is the first phase of the compiler. The lexical analyzer scans the source code and converts it into *tokens*. Tokens are small pieces of the program, like keywords (int), identifiers (x), numbers (10), or symbols (+, \*). Below are the tokens handled in this version of the compiler:

enum TokenType {  
PLUS,

    MINUS,

    MULTIPLY,

    DIVIDE,

    MODULO,

    ASSIGN,

    EQUAL,

    NOT\_EQUAL,

    LESS\_THAN,

    GREATER\_THAN,

    LESS\_EQUAL,

    GREATER\_EQUAL,

    IF,

    ELSE,

    WHILE,

    FOR,

    RETURN,

    INT,

    FLOAT,

    DOUBLE,

    CHAR,

    STRING,

    VOID,

    IDENTIFIER,

    INTEGER\_LITERAL,

    FLOAT\_LITERAL,

    STRING\_LITERAL,

    SEMICOLON,

    COMMA,

    LEFT\_PAREN,

    RIGHT\_PAREN,

    LEFT\_BRACE,

    RIGHT\_BRACE,

    LOGICAL\_AND,

    LOGICAL\_OR,

    LOGICAL\_NOT,

    END\_OF\_FILE,

    UNKNOWN

**Code Explanation**:

The code below performs the task of breaking an input source code into manageable bits called tokens. In very simple words, it scans the code character by character and groups them into meaningful units, which include number, keywords such as int, if, else etc., operators such as +, -, symbols such as (,).

Here are the key parts:

• It runs through the source code character by character, src checking whether it is a number, keyword, or operator.

• If it's a number, it calls consumeNumber() to process it and if it's a keyword such as int or if, then it assigns the proper token type T\_INT, T\_IF etc.

• If the character is an operator like +, -, or \* it adds the appropriate token.

class *Lexer*

{

private:

*string* source;

*size\_t* current;

    int line, column;

    vector<*Token*> tokens;

public:

*SymbolTable* symbolTable;

*string* currentScope = "global";

    Lexer(*const* *string* *&*src) : source(src), current(0), line(1), column(1) {}

    vector<*Token*> tokenize()

    {

        tokens.clear();

        while (!isAtEnd())

        {

            char c = advance();

            if (isspace(c))

            {

                if (c == '\n')

                {

                    line++;

                    column = 1;

                }

                continue;

            }

            switch (c)

            {

            case '+':

                tokens.push\_back(createToken(PLUS, "+"));

                break;

            case '-':

                tokens.push\_back(createToken(MINUS, "-"));

                break;

            case '\*':

                tokens.push\_back(createToken(MULTIPLY, "\*"));

                break;

            case '/':

                tokens.push\_back(createToken(DIVIDE, "/"));

                break;

            case '%':

                tokens.push\_back(createToken(MODULO, "%"));

                break;

            case '=':

                tokens.push\_back(match('=') ? createToken(EQUAL, "==") : createToken(ASSIGN, "="));

                break;

            case '<':

                tokens.push\_back(match('=') ? createToken(LESS\_EQUAL, "<=") : createToken(LESS\_THAN, "<"));

                break;

            case '>':

                tokens.push\_back(match('=') ? createToken(GREATER\_EQUAL, ">=") : createToken(GREATER\_THAN, ">"));

                break;

            case '!':

                tokens.push\_back(match('=') ? createToken(NOT\_EQUAL, "!=") : createToken(LOGICAL\_NOT, "!"));

                break;

            case '(':

                tokens.push\_back(createToken(LEFT\_PAREN, "("));

                break;

            case ')':

                tokens.push\_back(createToken(RIGHT\_PAREN, ")"));

                break;

            case '{':

                tokens.push\_back(createToken(LEFT\_BRACE, "{"));

                break;

            case '}':

                tokens.push\_back(createToken(RIGHT\_BRACE, "}"));

                break;

            case ';':

                tokens.push\_back(createToken(SEMICOLON, ";"));

                break;

            case ',':

                tokens.push\_back(createToken(COMMA, ","));

                break;

            case '"':

                tokens.push\_back(tokenizeStringLiteral());

                break;

            default:

                if (isdigit(c))

                {

                    tokens.push\_back(tokenizeNumber(c));

                }

                else if (isalpha(c) || c == '\_')

                {

                    tokens.push\_back(tokenizeIdentifierOrKeyword(c));

                }

                else

                {

                    tokens.push\_back(createToken(UNKNOWN, *string*(1, c)));

                }

            }

        }

        tokens.push\_back({END\_OF\_FILE, "", line, column});

        return tokens;

    }

## 02: Syntax Analysis

This phase checks whether the sequence of tokens follows the grammar rules of the programming language. It also builds a structured representation of the code, which is often called an Abstract Syntax Tree (AST). In this phase, the symbol table is used to record details about variables (names, types, etc.).

**Symbol Table Use Case**: The parser uses the symbol table to store information about variables declared in the code and ensures no duplicate declarations.

**Code DEscription**:

1. This code is part of the Syntactic Analyzer phase, which checks whether the sequence of tokens produced by the Lexical Analyzer follows the correct syntax of the language. It focuses on ensuring the structure of the program is valid according to the rules of the language. Since the entire code is too long, we will focus on one aspect, which is assignment, for better understanding.
2. **parseProgram()**:

Loops through the tokens until it reaches the end of the file (T\_EOF), calling parseStatement() for each statementt.

1. **parseStatement()**:

Identifies the type of statement (declaration, assignment, if, return, or block) and calls the appropriate parsing function like parseDeclaration() and parseAssignment(), etc.

If the token doesn’t match any known statement types, it reports a syntax error.

1. **parseAssignment()**:

* This function handles the assignment statements.
* First, it extracts the variable name using expectAndReturnValue(T\_ID) to ensure that the token is an identifier (the variable being assigned).
* Then, it checks if the variable has been declared in the symbol table by calling symTable.getVariableType(varName). This ensures the variable exists in the current scope.
* Next, it expects and processes the assignment operator (T\_ASSIGN), and parses the right-hand side expression using parseExpression().
* Finally, it expects a semicolon (T\_SEMICOLON) to complete the assignment statement.

This ensures the syntax of the program is correct and prepares it for the next phase.

private:

    bool isAtEnd() *const*

    {

        return current >= source.length();

    }

    char advance()

    {

        column++;

        return source[current++];

    }

    bool match(char expected)

    {

        if (isAtEnd() || source[current] != expected)

            return false;

        current++;

        column++;

        return true;

    }

*Token* createToken(*TokenType* type, *const* *string* *&*value = "")

    {

        return {type, value, line, column};

    }

*Token* tokenizeNumber(char first)

    {

*string* number(1, first);

        while (!isAtEnd() && isdigit(source[current]))

        {

            number += advance();

        }

        if (!isAtEnd() && source[current] == '.')

        {

            number += advance();

            while (!isAtEnd() && isdigit(source[current]))

            {

                number += advance();

            }

            return {FLOAT\_LITERAL, number, line, column};

        }

        return {INTEGER\_LITERAL, number, line, column};

    }

*Token* tokenizeIdentifierOrKeyword(char first)

    {

*string* identifier(1, first);

        while (!isAtEnd() && (isalnum(source[current]) || source[current] == '\_'))

        {

            identifier += advance();

        }

*static* *const* unordered\_map<*string*, *TokenType*> keywords = {

            {"if", IF}, {"else", ELSE}, {"while", WHILE}, {"for", FOR}, {"return", RETURN}, {"int", INT}, {"float", FLOAT}, {"double", DOUBLE}, {"char", CHAR}, {"string", STRING}, {"void", VOID}};

        auto it = keywords.find(identifier);

        if (it != keywords.end())

        {

            return {it->second, identifier, line, column};

        }

        else

        {

            if (!tokens.empty() && (tokens.back().type == INT || tokens.back().type == FLOAT ||

                                    tokens.back().type == DOUBLE || tokens.back().type == CHAR ||

                                    tokens.back().type == STRING || tokens.back().type == VOID))

            {

*string* type = tokens.back().value;

                symbolTable.addEntry(identifier, type, currentScope, line);

            }

            return {IDENTIFIER, identifier, line, column};

        }

    }

*Token* tokenizeStringLiteral()

    {

*string* literal;

        while (!isAtEnd() && source[current] != '"')

        {

            literal += advance();

        }

        if (!isAtEnd())

            advance(); *// Consume closing quote*

        return {STRING\_LITERAL, literal, line, column};

    }

};

## 03: Semantic Analysis

**Description**: Semantic analysis ensures that the program is meaningful. For example, it checks if variables are used without being declared or if types are mismatched (e.g., assigning a string to an integer variable).

* Ensure variables are declared before use.
* Type-checking (e.g., integers cannot be added to strings).

## 04: Intermediate Code Generation

In this phase, the compiler generates intermediate instructions that are simple and independent of the machine architecture. These instructions are easier to translate into assembly code. The intermediate code uses three-address code, where each instruction typically involves three operands: a destination, a source, and a result. This makes the code more flexible and machine-independent.

class *IntermediateCodeGenerator*

{

public:

    struct *ThreeAddressCode*

    {

*string* op;

*string* arg1;

*string* arg2;

*string* result;

*string* toString() *const*

        {

            if (arg2.empty())

            {

                return result + " = " + arg1;

            }

            return result + " = " + arg1 + " " + op + " " + arg2;

        }

    };

    vector<*ThreeAddressCode*> generate(*const* vector<*Token*> *&*tokens)

    {

        vector<*ThreeAddressCode*> intermediateCode;

*// Simple intermediate code generation based on tokens*

        for (*size\_t* i = 0; i < tokens.size(); ++i)

        {

            if (tokens[i].type == IDENTIFIER &&

                i + 1 < tokens.size() && tokens[i + 1].type == ASSIGN)

            {

*// Assignment handling*

                if (i + 2 < tokens.size())

                {

                    intermediateCode.push\_back({"=",

                                                tokens[i + 2].value,

                                                "",

                                                tokens[i].value});

                }

            }

*// Basic arithmetic operation handling*

            if (tokens[i].type == PLUS || tokens[i].type == MINUS ||

                tokens[i].type == MULTIPLY || tokens[i].type == DIVIDE)

            {

                if (i > 0 && i + 1 < tokens.size())

                {

                    intermediateCode.push\_back({tokenTypeToString(tokens[i].type),

                                                tokens[i - 1].value,

                                                tokens[i + 1].value,

                                                "temp" + to\_string(intermediateCode.size())});

                }

            }

        }

        return intermediateCode;

    }

private:

*string* tokenTypeToString(*TokenType* type)

    {

        switch (type)

        {

        case PLUS:

            return "+";

        case MINUS:

            return "-";

        case MULTIPLY:

            return "\*";

        case DIVIDE:

            return "/";

        default:

            return "?";

        }

    }

};

## Phase 5: Code Generation

The final phase converts intermediate instructions into assembly code. Assembly code is machine-readable and can be executed by the processor. In this case, **RISC-V** assembly language is used for the conversion. RISC-V is a popular open-source instruction set architecture (ISA) that is simple and flexible, making it ideal for educational purposes and custom processor designs. The intermediate instructions generated earlier are mapped to corresponding RISC-V instructions, ensuring that the generated code can be efficiently executed on a RISC-V processor.

Below code generates RISC-V of any arithmetic operation involving only two operands, which is ensured by using three address code in earlier phase. This function processes binary operations (like addition, subtraction, multiplication, or division) in a line of intermediate code.

class *AssemblyGenerator*

{

public:

*string* generate(*const* vector<*IntermediateCodeGenerator*::*ThreeAddressCode*> *&*intermediateCode)

    {

*stringstream* assembly;

        assembly << ".intel\_syntax noprefix\n";

        assembly << ".global main\n\n";

        assembly << "main:\n";

        assembly << "    push rbp\n";

        assembly << "    mov rbp, rsp\n\n";

        for (*const* auto *&*code : intermediateCode)

        {

            generateInstruction(code, assembly);

        }

        assembly << "    mov rax, 0\n";

        assembly << "    leave\n";

        assembly << "    ret\n";

        return assembly.str();

    }

private:

    void generateInstruction(

*const* *IntermediateCodeGenerator*::*ThreeAddressCode* *&*code,

*stringstream* *&*assembly)

    {

        if (code.op == "=")

        {

            assembly << "    # Assignment\n";

            assembly << "    mov rax, " << code.arg1 << "\n";

            assembly << "    mov [" << code.result << "], rax\n";

        }

        else if (code.op == "+")

        {

            assembly << "    # Addition\n";

            assembly << "    mov rax, " << code.arg1 << "\n";

            assembly << "    add rax, " << code.arg2 << "\n";

            assembly << "    mov [" << code.result << "], rax\n";

        }

        else if (code.op == "-")

        {

            assembly << "    # Subtraction\n";

            assembly << "    mov rax, " << code.arg1 << "\n";

            assembly << "    sub rax, " << code.arg2 << "\n";

            assembly << "    mov [" << code.result << "], rax\n";

        }

        else if (code.op == "\*")

        {

            assembly << "    # Multiplication\n";

            assembly << "    mov rax, " << code.arg1 << "\n";

            assembly << "    imul rax, " << code.arg2 << "\n";

            assembly << "    mov [" << code.result << "], rax\n";

        }

        else if (code.op == "/")

        {

            assembly << "    # Division\n";

            assembly << "    mov rax, " << code.arg1 << "\n";

            assembly << "    mov rbx, " << code.arg2 << "\n";

            assembly << "    div rbx\n";

            assembly << "    mov [" << code.result << "], rax\n";

        }

    }

};

*// Compiler Class*

class *Compiler*

{

public:

    void compile(*const* *string* *&*sourceCode)

    {

*Lexer* lexer(sourceCode);

        vector<*Token*> tokens = lexer.tokenize();

*// Print tokens*

        cout << "Tokens:\n";

        for (*const* auto *&*token : tokens)

        {

            cout << token.toString() << endl;

        }

*// Print Symbol Table*

        lexer.symbolTable.print();

*// Intermediate Code Generation*

*IntermediateCodeGenerator* intermediateGenerator;

        auto intermediateCode = intermediateGenerator.generate(tokens);

*// Print Intermediate Code*

        cout << "\nIntermediate Code:\n";

        for (*const* auto *&*code : intermediateCode)

        {

            cout << code.toString() << endl;

        }

*// Assembly Code Generation*

*AssemblyGenerator* assemblyGenerator;

*string* assemblyCode = assemblyGenerator.generate(intermediateCode);

*// Print Assembly Code*

        cout << "\nAssembly Code:\n";

        cout << assemblyCode << endl;

    }

};

# More Features:

Below are the additional features I added:

1. Supporting more variable types for declaration like double, bool and implement more features like functions, there parameters etc.
2. Handle conditional Statements, if else.
3. Handling loops both for and while.
4. Generating RISC-V assembly code from Three Address code (intermediate Code).