Compiler Construction

Project Supervisor

Mr. Laeeq Khan Niazi

Submitted By

Uswa Arif 2021-CS-77

Department of Computer Science

University of Engineering and Technology, Lahore

Pakistan

Contents

Chapter 1

Introduction

The task of building a compiler involves a series of complex steps aimed at translating high-level source code into executable machine code. Compilers are crucial tools in modern programming environments, as they enable the conversion of human-readable code into a form that a computer can understand. This project focuses on constructing a simple compiler in C++ that processes a user-defined language supporting a variety of features, including primitive data types, arithmetic operations, logical operators, conditional statements, and loop structures. The design follows the standard compiler phases, from lexical analysis (tokenization) to syntax analysis (parsing), intermediate code generation, and ultimately the production of assembly code.

The first key phase in the compiler construction is Tokenization, where the raw source code is divided into meaningful units, called tokens. These tokens are essential for understanding the structure and meaning of the code. In this compiler, a set of predefined token types represents different constructs of the language, such as integers, strings, floats, and characters, as well as operators like addition, subtraction, division, multiplication, and comparison. The Lexer component is responsible for scanning the source code and classifying these tokens, which are then passed to the Parser for syntactic analysis. The parser ensures that the tokens are arranged in a manner that follows the grammar of the language, checking for correct usage and creating an Intermediate Representation.

The Symbol Table plays an integral role throughout the compilation process by storing information about variables and other identifiers encountered in the source code. This table allows the compiler to track the scope and usage of different entities, such as integers, floats, or strings, and helps to catch errors related to undeclared variables or type mismatches. Following the parsing stage, the compiler generates Intermediate Code in the form of three-address code (TAC), which serves as an abstraction of the program's logic. This intermediate representation is easier to optimize and transform into machine code. Finally, the Assembly Code Generation phase produces the final code that can be executed on the target machine.

An important feature of the compiler is its ability to handle errors gracefully. It includes mechanisms to report errors with detailed information, such as line and column numbers, ensuring that the programmer can easily identify the issues in the source code. The error handling system covers syntax errors, type mismatches, undeclared variables, and other common mistakes that programmers make during development.

The compiler also supports external file execution, where users can input the name of a C++ source code file, and the compiler will compile the code into executable assembly code. This feature enhances usability and makes the compiler more adaptable for real-world scenarios.

1.1 Scope of the Project

The scope of this compiler construction project involves creating a basic compiler for a custom programming language using C++. The primary goal is to provide support for essential language features such as data types, arithmetic and logical operations, conditional statements, and loop structures, with a focus on practical use cases in software development. The project aims to create a fully functional compiler capable of handling source code written in a simple language, transforming it into executable machine code.

The compiler follows standard phases in compilation, including tokenization, lexical analysis, parsing, intermediate code generation, and assembly code generation. It is designed to work with a variety of primitive data types, including int, float, char, and string, as well as control flow statements such as if, else, and while loops. The project also provides robust error handling, reporting errors with line and column numbers to help developers troubleshoot issues in their code. Additionally, the compiler is built to handle multiple token types that include variables, operators, keywords, and symbols, while ensuring that the generated assembly code can be executed by the machine.

1.2 Overview of the Document

This document provides a detailed explanation of the implementation and design of a C++ compiler that processes custom programming language code. The compiler follows a structured approach, breaking down the process into several key phases: Tokenization, Lexical Analysis, Parsing, Intermediate Code Generation, and Assembly Code Generation. Each of these phases plays a critical role in transforming the high-level code into a format suitable for execution on a computer.

The document also highlights the features and capabilities of the compiler, including the support for various data types, operators, and control flow constructs. Furthermore, it discusses the error handling mechanisms implemented to ensure robust debugging, with precise error reporting indicating the line and column number. The document covers the tokenization and parsing process, as well as the generation of intermediate and assembly code. Lastly, it emphasizes the extra features and advanced functionalities added to the compiler to enhance its versatility and performance.

1.3 Features Implemented in the Compiler

The implemented compiler supports a wide range of language features, both in terms of data types and operators, which make it a versatile tool for compiling programs written in the designed language. The following core features have been implemented:

1.3.1 Data Types

- **int:** Integer type for whole numbers.
- **char:** Character type for single characters.
- **float:** Floating-point type for decimal numbers.
- **string:** String type for sequences of characters.

1.3.2 Operators

- **Arithmetic Operators:**
  PLUS (+), MINUS (-), MUL (*), DIV (/) for addition, subtraction, multiplication, and division.
- **Comparison Operators:**
  LT (<), GT (>), LE (<=), GE (>=), ET (==) for less than, greater than, less than or equal to, greater than or equal to, and equal to comparisons.
- **Logical Operators:**
  AND, OR, NOT for logical conjunction, disjunction, and negation.

- **Conditional Statements:**
  IF, ELSE for conditional branching in code execution.
- **Additional Features:**
  AGAR (a custom operator in the language), used for operations similar to the IF condition.

1.3.3 Other Token Types

A variety of tokens are supported, enabling the identification and processing of different elements in the source code:

- The supported tokens are Identifiers, Numbers, Return keyword, Assign(equal to), Left Parenthesis, Right Parenthesis, Left Bracket, Right Bracket, Semicolon, End of File. The keywords are: T_ID, T_NUM, T_RETURN, T_ASSIGN, T_LPAREN, T_RPAREN, T_LBRACE, T_RBRACE, T_SEMICOLON, T_EOF.

1.3.4 Symbol Table

A symbol table is maintained to store information about variables, identifiers and their respective data types. This ensures that variables are declared before use and type consistency is maintained.

1.3.5 Intermediate Code Generation

The compiler produces three-address code (TAC) as an intermediate representation of the program. This simplified code structure makes it easier to perform optimizations before generating assembly code.

1.3.6 Assembly Code Generation

The final phase of the compiler generates assembly code, which can be directly executed by a computer.

1.4 Error Handling

The compiler includes an advanced error handling system that provides clear and informative error messages. These messages include line and column numbers, making it easy to pinpoint where the error occurred in the code. Common errors such as undeclared variables, type mismatches, syntax errors, and invalid expressions are all caught and reported.

1.5 Extra Features

The following extra features have been added to the compiler:

1.5.1 Extended Token Types

In addition to the basic token types like T_INT, T_NUM, and T_ID, the compiler also supports extra operators and keywords such as T_FOR, T_WHILE, T_FLOAT, T_STRING, T_AGAR, T_CHAR, among others. These tokens enable the compiler to handle a wide range of language constructs.

1.5.2 Enhanced Operators

The support for various other operators, including comparison (LT, GE, ET), and logical (AND, OR, NOT), allows the compiler to handle a wide variety of expressions and conditions in the source code.

1.5.3 File-Based Execution

The compiler is designed to take a filename as input during execution. By running a command like compiler.exe code.cpp, the compiler processes the provided C++ source code file, performs lexical analysis, parses it, generates intermediate code, and outputs the corresponding assembly code. This feature makes it easy to compile multiple files in a batch process.

1.6 Technology Stack and IDEs

- **Programming Language:** C++
- **Compiler:** C++
- **IDE:** VS Code

1.7 Phases of the Compile

The phases of the compiler refer to the different stages through which the input program passes to be transformed into executable code. Each phase is responsible for a specific task, ensuring that the program is analyzed, optimized, and converted into machine-readable instructions. Below is a breakdown of the key phases in the compilation process:

- Tokenization (Scanner)
- Lexical Analysis (Lexer)
- Parsing
- Intermediate Code Generation
- Assembly Code Generation

Chapter 2

Phase 1: Tokenization

Tokenization is one of the foundational phases of a compiler, where the input source code is divided into smaller units called tokens. These tokens represent meaningful sequences of characters in the source code, such as keywords, operators, identifiers, literals, and symbols. The primary goal of tokenization is to break the input code into manageable pieces that can be easily understood and processed by the subsequent stages of the compiler, such as the parser. The process involves recognizing different components of the code and categorizing them into predefined types, which are later used for syntactical and semantic analysis.

2.1 ENUM TokenType

In my compiler implementation, I have used an enum TokenType to represent the various types of tokens that can be encountered in a C++ program. An enum, or enumeration, is a user-defined data type that consists of a set of named values, or elements, that are used to represent a fixed set of constants. This enum serves as a set of constants, each corresponding to a distinct type of token.
The Tokens Types are:

- **T_INT** represents integer datatype used to declare variables that store integer values.
- **T_ID** represents Identifiers. An identifier is used to name variables, functions, arrays, etc.
- **T_NUM** represents a literal number in the code, typically an integer or floating-point number. This token captures numeric constants in the source code.
- **T_IF** represents the if keyword used for conditional statements. It is used to check a condition and execute a block of code if the condition is true.
- **T_ELSE** represents the else keyword used in conjunction with if to define an alternative block of code that is executed if the condition of the if statement is false.
- **T_RETURN** represents the return keyword used to return a value from a function. It indicates the end of a method's execution.
- **T_ASSIGN** represents the assignment operator = used to assign values to variables.
- **T_PLUS** represents the addition operator +. It is used to add two values or variables.
- **T_MINUS** represents the subtraction operator -. It is used to subtract one value from another.
- **T_MUL** represents the multiplication operator *. It is used to multiply two values.
- **T_DIV** represents the division operator /. It is used to divide one value by another.

- **T_LPAREN** represents the left parenthesis (. Parentheses are used for grouping expressions.
- **T_RPAREN** represents the right parenthesis). It is used to close a grouping expression.
- **T_LBRACE** represents the left brace . Braces are used to define a block of code, such as in loops, and conditionals.
- **T_RBRACE** represents the right brace . It is used to close a block of code that was opened with {.
- **T_SEMICOLON** represents the semicolon ;, which is used to terminate statements in many programming languages.
- **T_GT** represents the "greater than" operator >. It is used to compare two values and check if one is greater than the other.
- **T_EOF** represents the end-of-file token. It marks the point where the lexer finishes processing the input.
- **T_FOR** represents the for keyword, which is used to start a for-loop for iterating over a range or collection.
- **T_WHILE** represents the while keyword, used to create a while-loop that continues as long as a condition is true.
- **T_EQ** represents the equality operator ==. It is used to compare two values to check if they are equal.
- **T_LE** represents the "less than or equal to" operator <=. It compares two values to check if the first is less than or equal to the second.
- **T_AND** represents the logical AND operator &&. It is used to combine two boolean expressions, returning true only if both are true.
- **T_FLOAT** represents the floating-point data type. This token is used for numbers with decimal points.
- **T_STRING** represents a string literal, which is typically enclosed in double quotes. It is used to represent textual data.
- **T_AGAR** represent a specific keyword in the language similar to the IF condition.
- **T_OR** represents the logical OR operator ||. It is used to combine two boolean expressions, returning true if at least one is true.
- **T_NOT** represents the logical NOT operator !. It negates a boolean expression, returning true if the expression is false and vice versa.
- **T_GE** represents the "greater than or equal to" operator >=. It compares two values to check if the first is greater than or equal to the second.
- **T_LT** represents the "less than" operator <. It is used to compare two values and check if the first is less than the second.
- **T_CHAR** represents the character data type. This is used for single characters, typically enclosed in single quotes.

2.2 Token struct

The struct Token, which encapsulates all the necessary information about a token. This Token struct is crucial for managing the tokens and their associated information in a way that can be easily passed along through the various phases of the compiler. It allows you to not only identify the type of each token but also track where it appeared in the source code. This structure holds the following fields:

- **type:** A field of type TokenType, which stores the type of the token. This corresponds to the different values in the TokenType enum (e.g., T_IF, T_ID, T_INT).

- **value:** A string that stores the actual value of the token.
- **lineNumber:** The line number in the source code where the token was found. This is crucial for error handling, as it helps pinpoint the location of any issues in the code.
- **columnNumber:** The column number within the line where the token starts. Similar to the line number, this helps in precise error reporting.
- **dataType:** An optional field (defaulted to an empty string), which can hold additional information like the data type of a variable (e.g., "int", "float") when needed.

Chapter 3

Phase 2: Lexical Analysis

Lexical analysis is the first phase of the compilation process, which involves breaking down a stream of characters (source code) into meaningful units called tokens. This phase identifies keywords, operators, identifiers, literals, and symbols, which are essential building blocks for the subsequent phases of parsing and code generation.

3.1 Lexer Class

The Lexer class in the code performs lexical analysis by reading through the source code and generating tokens based on the current character or sequence of characters. Here's a breakdown of each phase in lexer implementation:

3.1.1 Class Initialization (Input Processing)

- **src:** The source code passed to the lexer.
- **pos:** The current position of the pointer in the source code.
- **line and column:** Track the line and column of the current character, which is useful for error reporting.

3.1.2 Methods for Consuming Tokens(Token Generation)

- **consumeNumber():**
  This method consumes both integer and floating-point numbers. It checks if the current character is a digit and continues reading until it encounters a non-digit character. If a decimal point (.) is found, it treats the number as a floating-point number. It returns the string representing the number (either integer or floating-point).
- **consumeString():**
  This method handles string literals enclosed in double quotes ("). It reads characters between the quotes until it reaches the closing quote. If the string is unterminated, an error message is shown, and the program exits.
- **consumeCharLiteral():**
  Similar to consumeString(), but it handles character literals enclosed in single quotes ('). It checks for invalid character literals and reports an error if found.
- **consumeWord():**
  This method consumes alphanumeric characters (which could form identifiers or keywords). It keeps consuming characters as long as they are alphanumeric.

3.1.3 Tokenization Process (tokenize Method)

The tokenize() method is the core of the lexer. It iterates over the source code and identifies tokens by checking the current character and performing appropriate actions. Here's a breakdown of its working:

- **Whitespace Handling:**
  If the current character is whitespace (isspace(current)), the lexer simply skips it. If the character is a newline ('\n'), it updates the line number and resets the column counter.
- **Comment Handling:**
  If a comment is encountered (//), the lexer skips all characters until the end of the line.
- **Numeric Literals:**
  If the character is a digit, the consumeNumber() method is called to extract the number, which is then added to the token list.
- **Identifiers and Keywords:**
  If the character is alphabetic, the consumeWord() method is used to extract the word. Depending on the word, it may be classified as a keyword (e.g., if, else, return) or an identifier (T_ID).
- **String Literals:**
  If the character is a double quote ("), the consumeString() method is invoked to extract the string literal.
- **Character Literals:**
  If the character is a single quote ('), the consumeCharLiteral() method is called to handle character literals.
- **Multi-Character Operators:**
  If characters like ==, <=, >=, &&, ||, or != are encountered, the lexer checks for the second character and creates a corresponding token
- **Single-Character Operators:**
  If a single operator like =, +, -, *, /, etc., is encountered, a corresponding token is added to the list.
- **End of File:**
  When the end of the file is reached, the lexer adds an EOF token to signify the end of the source code.

3.1.4 Syntax Error Handling

If the lexer encounters an unexpected character or an invalid literal (like an unterminated string or an invalid character literal), it prints an error message with the line and column number and then exits the program.

3.1.5 Final Token List

The lexer returns a list of tokens representing the source code, which can then be passed to the parser for further analysis.

Chapter 4

Phase 3: Parser

In a compiler, the parser is responsible for analyzing the syntax of a given sequence of tokens (which are generated by the lexical analyzer or scanner). The parser checks whether the tokens follow the correct syntax rules of the programming language. It transforms the sequence of tokens into a parse tree, which represents the syntactic structure of the source code.

The parser also detects syntax errors (when the code violates the grammar rules) and can trigger semantic checks. In this context, the parser plays a crucial role in ensuring that the code follows both syntactic and semantic correctness.

In my code, the Parser class handles the parsing of a program's syntax, processes the tokens, and invokes various methods based on different types of statements (such as declarations, assignments, loops, and conditionals). Additionally, the parser ensures semantic correctness by checking for issues such as redeclared variables or type mismatches.

4.1 Parser Class

The Parser class is responsible for parsing the sequence of tokens generated by the lexical analyzer and transforming them into a more structured representation (an Intermediate Code). The class not only handles syntax but also checks for semantic correctness (like undeclared variables or type mismatches) and generates intermediate code as it parses.

4.1.1 Class Initialization

- **tokens:** A vector containing the list of tokens produced by the lexical analyzer.
- **symTable:** A reference to the symbol table, which stores information about variables.
- **pos:** This is a position pointer that tracks the current token being processed from the list of tokens.
- **icg:** A reference to the intermediate code generator that produces intermediate code during parsing.

4.1.2 Methods in Parser Class

- **parseProgram():**
  The parseProgram method acts as the entry point for the parser. It processes the tokens sequentially, calling the appropriate methods for different statements until the end of the file(EOF) is reached. The

method loops through all the tokens, calling parseStatement() for each token to parse individual statements. Once all tokens are consumed, it indicates that parsing is completed successfully. The method also prints the final symbol table.

- **parseStatement():**
  The parseStatement method is responsible for identifying and handling different types of statements in the source code. It checks the type of token encountered (e.g., variable declarations, assignments, conditionals, loops) and delegates parsing to the appropriate method based on the token type. If an unexpected token is encountered, the method raises a syntax error and exits the program.
- **parseDeclaration():**
  The parseDeclaration method handles variable declarations, ensuring that the variables are declared correctly and inserted into the symbol table. It checks for valid data types (such as int, float, string, etc.) and ensures that the variable name is valid. If the variable is already declared, a semantic error is raised. The method also inserts the variable into the symbol table for future use, associating it with its data type.
- **parseAssignment():**
  The parseAssignment method parses assignment statements, ensuring that the left-hand side variable is declared and the right-hand side expression is valid. The method parses the right-hand side of the assignment as an expression and generates intermediate code for the assignment operation. It handles the assignment process, ensuring both sides of the assignment are syntactically correct.
- **parseIfStatement():**
  The parseIfStatement method parses if statements, including optional else clauses. It expects a condition inside parentheses and processes it as an expression. The true branch (code block) is parsed, and if an else clause exists, it is also parsed. It does not parse else-if block.
- **parseAgarStatement():**
  It works similar to the If condition statement but it does not have any else-block.
- **parseReturnStatement():**
  The parseReturnStatement method handles parsing the return statement in the source code.
- **parseBlock():**
  The parseBlock method parses a block of code, often enclosed in curly braces . A block typically contains multiple statements and is treated as a unit in programming languages. The method starts by checking for an opening curly brace to mark the start of the block. It then enters a loop, parsing statements one by one within the block using the parseStatement() method. The method continues parsing until it encounters a closing curly brace or runs out of statements. Any missing braces or syntax errors within the block will result in an error.
- **parseExpression():**
  The parseExpression method is responsible for parsing an expression, which could involve variables, constants, or more complex operations (such as addition, subtraction, multiplication, etc.). It calls lower-level parsing methods (such as parseTerm() and parseFactor()) to handle the components of the expression. The method uses precedence and associativity rules to correctly parse expressions with multiple operators (e.g., handling operator precedence between addition and multiplication). After parsing the full expression, the method returns the parsed representation of the expression.
- **parseForLoop():**
  The parseForLoop method is used to parse for loop constructs, handling the initialization, condition, update, and body.
- **parseWhileStatement():**
  The parseWhileStatement method handles while loops, ensuring that the condition is valid and the loop body is processed correctly. It expects a condition inside parentheses and processes it as an expression.

- **parseTerm():**
  The parseTerm method is responsible for parsing a term, which typically involves parsing factors (variables, constants, or parenthesized expressions) and handling operators like multiplication and division.
- **parseFactor():**
  The parseFactor method is responsible for parsing a factor, which could be a literal, a variable, or a parenthesized expression. The method checks if the current token is a literal (e.g., a number or string). If the token is a variable, it verifies that the variable has been declared and adds it to the intermediate representation. The method also checks for parentheses to handle expressions inside parentheses, ensuring proper precedence by recursively calling parseExpression for the expression within the parentheses. If the token is none of these, it raises a syntax error.
- **expect():**
  The expect method is used to check whether the current token matches the expected token type. If the types do not match, it raises a syntax error with a descriptive message and exits the program. If the expected token is found, the method consumes it by advancing the position pointer to the next token.
- **expectAndReturnValue():**
  The expectAndReturnValue method is a utility function that checks if the current token matches the expected token type and returns the value of that token if it matches.

4.1.3 Error Handling in the Parser

Error handling in the parser is critical to ensure the source code is syntactically correct. If any syntax or semantic error is found, the parser will stop processing and print an error message.

- **Syntax Errors** The parser checks for syntax errors such as mismatched parentheses or missing tokens (e.g., expecting a semicolon after a statement).
- **Semantic Errors** The parser also handles semantic errors, such as redeclaring a variable or using an undeclared variable.

4.2 Symbol Table in the Parser Phase

The Symbol Table is a critical data structure in the Parser Phase of a compiler, used to keep track of all identifiers (such as variables, functions, classes, etc.) encountered in the source code. It stores relevant information about these identifiers, including their type. The symbol table is primarily used for two purposes during parsing:

- **Symbol Lookup**
- **Semantic Validation**

During the parsing phase, every time a new identifier is encountered (e.g., when parsing a variable or function), it is added to the symbol table. If the identifier already exists, an error is raised.

4.2.1 Methods in the Symbol Table

- **Insert:** The insert method adds a new symbol to the symbol table.
- **Lookup:** The lookup method checks if a symbol exists in the symbol table. It takes the symbol name as an argument and returns the associated data if the symbol is found; otherwise, it returns None.
- **Duplicate Check:** The check_duplicate method checks if a symbol already exists in the symbol table, helping to detect redeclarations of variables or functions in the same scope.

Chapter 5

Phase 4: Intermediate Code Generator

The Intermediate Code Generator is a phase in a compiler that bridges the gap between the source code and the target code (such as assembly or machine code). The intermediate code is typically more abstract than machine code but lower-level than source code, allowing the compiler to optimize it before translating it to assembly or machine code. One common form of intermediate code is Three-Address Code (TAC), which breaks down expressions into simpler components, using at most three operands.

In this implementation of the Intermediate Code Generator, it focuses on generating TAC from higher-level code. The generated intermediate code will be in the form of simple instructions that can later be translated to assembly code.

5.1 Three-Address Code (TAC)

Three-Address Code (TAC) is a type of intermediate code commonly used in compilers. It represents code in a simple form where each instruction consists of at most three addresses (operands), making it easier to generate target code such as assembly. The addresses refer to variables, constants, or temporary variables generated during the translation of complex expressions.
TAC typically uses the following formats:

- **Assignment:** It represents format like "x = y" means assigns the value of y to x.
- **Binary Operations:** It represents format "z = x operation y". A binary operation (addition, subtraction, multiplication, division) between y and z, with the result stored in x.
- **Unary Operations:** It represents format "x = operation y".
- **Conditional Jumps:** It represents format "if x op y goto L". If the condition x op y is true, the program jumps to label L.
- **Unconditional Jumps:** It represents format "goto L". Directly jumps to label L without any condition.
- **Labels:** It represents format "L". It marks a specific location in the code that can be referenced by jump instructions.

5.2 Intermediate Code Generator Class

5.2.1 Class Initialization

- **instructions:** A vector of strings that holds the list of generated three-address code instructions.

- **tempCount:** A counter to keep track of the temporary variables (e.g., t1, t2, etc.) used in the intermediate code.

5.2.2 Class Methods

- **newTemp():**
  The function newTemp() is responsible for generating new temporary variables (like t1, t2, etc.) as needed. These variables store the intermediate results of expressions.
- **addInstruction():**
  This method adds a new instruction (in the form of a string) to the list of instructions.
- **printInstructions():**
  This method outputs the intermediate code (TAC) to the console for inspection or further processing.

Chapter 6

Assembly Code Generation

The assembly code generation phase in a compiler converts intermediate code, such as Three-Address Code (TAC), into low-level assembly instructions specific to the target machine. This phase bridges the gap between high-level language constructs and machine instructions, ensuring that operations and variables are mapped correctly to the architecture's registers and memory.

Key Objectives

The code generation phase focuses on translating the intermediate representation into target machine code or assembly language. The key objectives include:

- **Register Allocation**: Mapping variables or temporary values to machine registers.
- **Instruction Mapping**: Translating high-level operations (e.g., arithmetic, logical operations) into corresponding assembly instructions.
- **Memory Management**: Managing values that don't fit into registers by storing them in memory and loading them when necessary.
- **Handling Control Flow**: Implementing conditional and unconditional jumps through assembly-level branch instructions.

6.1 Keywords in Assembly Code

The assembly code have following keywords:

- **LOAD** load the operand into a register.
- **STORE** store the result into a temporary register or memory.
- **ADD** Add two operands.
- **MUL** Multiply two operands.
- **DIV** Divide two operands.
- **NEG** Negative operand
- **COMP** Compare two operands.
- **BRANCH** Branch based on the comparison result (e.g., BLT, BEQ).
- **JMP** Jump to the code.

6.2 Operations in Assembly Code

6.2.1 Arithmetic Operations

The key steps are:
1- LOAD the operand into a register.
2- Perform the arithmetic operation (e.g., ADD) between the operands.
3- STORE the result into a temporary register or memory.

6.2.2 Assignment Operations

The key steps are:
1- LOAD the value of the variable being assigned into a register.
2- STORE the register's value into the target variable.

6.2.3 Unary Operations

The key steps are:
1- LOAD the value of the variable being assigned into a register.
2- Unary operations (like negation) are also handled by loading the operand, applying the unary operator (e.g., NEG).
3- STORE the register's value into the target variable.

6.2.4 Conditional Jumps

The key steps are:
1- LOAD the first operand.
2- COMPare the two operands.
3- BRANCH based on the comparison result (e.g., BLT, BEQ).

6.2.5 UnConditional Jumps

Unconditional jumps (goto L2) are translated into assembly as a simple JMP instruction..

6.2.6 Labels

Labels in TAC are directly carried over to assembly.

Chapter 7

Complete Code

```cpp

#include<iostream>
#include<vector>
#include<string>
#include<cctype>
#include<map>
#include <unordered_map>
#include <fstream>

using namespace std;

enum TokenType
{
    T_INT, T_ID, T_NUM, T_IF, T_ELSE, T_RETURN,
    T_ASSIGN, T_PLUS, T_MINUS, T_MUL, T_DIV,
    T_LPAREN, T_RPAREN, T_LBRACE, T_RBRACE,
    T_SEMICOLON, T_GT, T_EOF,
    T_FOR, T_WHILE, T_EQ, T_LE, T_AND, T_FLOAT, T_STRING,
    T_AGAR, T_OR, T_NOT, T_GE, T_LT, T_CHAR
};



struct Token
{
    TokenType type;
    string value;
    int lineNumber;
    int columnNumber;
    string dataType;

    Token(TokenType t, const string& val, int ln, int col, const string& dt = "")
        : type(t), value(val), lineNumber(ln), columnNumber(col), dataType(dt) {}
};

// Symbol Table Class using unordered_map
class SymbolTable {
    private:
        map<string, string> table;
    public:
        void insert(const string& name, const string& type) {
            if (table.find(name) != table.end())
            {
                throw runtime_error("Semantic␣error:␣Variable␣'" + name + "'␣is␣already␣
                    declared.");
            }
            table[name] = type;
        }

        string lookup(const string& name) {
            if (table.find(name) != table.end()) {
                return table[name];
            }
            return "";
        }

        void display() {
            cout << "\nSymbol Table:\n";
            for (const auto& entry : table) {
                cout << "Name: " << entry.first << ", Type: " << entry.second << endl;
            }
        }

        bool isDeclared(const string &name) const
        {
            return table.find(name) != table.end();
        }
};

class IntermediateCodeGnerator {
    public:
        vector<string> instructions;
        int tempCount = 0;

        string newTemp() {
            return "t" + to_string(tempCount++);
        }

        void addInstruction(const string &instr) {
            instructions.push_back(instr);
        }

        void printInstructions() {
            cout << "Three-Address Code (TAC):" << endl;
            for (const auto &instr : instructions) {
                cout << instr << endl;
            }
        }

        // Translate Three-Address Code (TAC) to Assembly code
        void generateAssemblyCode() {
        unordered_map<string, string> tempToRegMap;  // Maps TAC variables to registers
        string nextReg = "R1";                       // Start with register R1
        int regCount = 1;                            // Register counter
        int labelCount = 1;                          // Label counter

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

        for (const auto &instr : instructions) {
            vector<string> parts = splitInstruction(instr);

            // Handle arithmetic operations (addition, subtraction, multiplication, division)
            if (parts.size() == 5 && (parts[3] == "+" || parts[3] == "-" || parts[3] == "*"
                || parts[3] == "/")) {
                string operation = parts[3];  // Get the operator
                string assemblyOp;

                // Map the operator to the corresponding assembly instruction
                if (operation == "+") assemblyOp = "ADD";
                else if (operation == "-") assemblyOp = "SUB";
                else if (operation == "*") assemblyOp = "MUL";
                else if (operation == "/") assemblyOp = "DIV";

                // Generate assembly for arithmetic operations
                cout << "LOAD␣" << parts[2] << ",␣" << nextReg << endl;
                cout << assemblyOp << "␣" << parts[4] << ",␣" << nextReg << endl;
                tempToRegMap[parts[0]] = nextReg;  // Store result in temp register
                cout << "STORE␣" << nextReg << ",␣" << parts[0] << endl;
            }

            // Handle assignments (e.g., a = b)
            else if (parts.size() == 3 && parts[1] == "=") {
                // TAC: a = b
                // Assembly: LOAD b, R1 -> STORE R1, a
                cout << "LOAD␣" << parts[2] << ",␣" << nextReg << endl;
                cout << "STORE␣" << nextReg << ",␣" << parts[0] << endl;
            }

            // Handle unary operations (e.g., t3 = -a)
            else if (parts.size() == 4 && parts[2] == "-") {
                // TAC: t3 = -a
                // Assembly: LOAD a, R1 -> NEG R1 -> STORE R1, t3
                cout << "LOAD␣" << parts[3] << ",␣" << nextReg << endl;
                cout << "NEG␣" << nextReg << endl;
                tempToRegMap[parts[0]] = nextReg;
                cout << "STORE␣" << nextReg << ",␣" << parts[0] << endl;
            }

            // Handle conditional jumps (e.g., if a < b goto L1)
            else if (parts.size() == 6 && parts[0] == "if") {
                string condOp = parts[2];  // Get comparison operator
                string assemblyCond;

                // Map the condition operator to assembly branch condition
                if (condOp == "<") assemblyCond = "BLT";
                else if (condOp == "<=") assemblyCond = "BLE";
                else if (condOp == ">") assemblyCond = "BGT";
                else if (condOp == ">=") assemblyCond = "BGE";
                else if (condOp == "==") assemblyCond = "BEQ";
                else if (condOp == "!=") assemblyCond = "BNE";

                // Assembly: LOAD a, R1 -> COMP b, R1 -> CONDITIONAL BRANCH L1
                cout << "LOAD␣" << parts[1] << ",␣" << nextReg << endl;
                cout << "COMP␣" << parts[3] << ",␣" << nextReg << endl;
                cout << assemblyCond << "␣" << parts[5] << endl;
            }

            // Handle unconditional jumps (e.g., goto L2)
            else if (parts.size() == 2 && parts[0] == "goto") {
                // TAC: goto L2
                // Assembly: JMP L2
                cout << "JMP␣" << parts[1] << endl;
            }

            // Handle labels (e.g., L1:)
            else if (parts.size() == 1 && parts[0].back() == ':') {
                // TAC: L1:
                // Assembly: L1:
                cout << parts[0] << endl;
            }

            // Update next available register
            nextReg = "R" + to_string(++regCount);
        }
    }

    private:
        // Helper function to split TAC instruction into parts
        vector<string> splitInstruction(const string &instr) {
            vector<string> parts;
            string part;
            for (char ch : instr) {
                if (isspace(ch)) {
                    if (!part.empty()) {
                        parts.push_back(part);
                        part.clear();
                    }
                } else {
                    part += ch;
                }
            }
            if (!part.empty()) parts.push_back(part);
            return parts;
        }
};



class Lexer{
    private:
        string src;   //code
        size_t pos;   //position of pointer
        int line;
        int column;


    public:
        Lexer(const string &src)
        {
            this->src = src;
            this->pos = 0;
            this->line = 1;
            this->column = 1;
        }

        string consumeNumber() {
            size_t start = pos;
            bool isFloat = false;

            while (pos < src.size() && isdigit(src[pos])) {
                pos++;
            }

            if (pos < src.size() && src[pos] == '.') {
                isFloat = true;
                pos++;

                while (pos < src.size() && isdigit(src[pos])) {
                    pos++;
                }
            }

            if (isFloat) {
                return src.substr(start, pos - start);  //Floating-point number
            } else {
                return src.substr(start, pos - start);  //Integer number
            }
        }

        string consumeString() {
            size_t start = pos;
            pos++;

            while (pos < src.size() && src[pos] != '"') {
                pos++;
            }

            if (pos < src.size() && src[pos] == '"') {
                pos++;
            } else {
                cout << "Error:␣Unterminated␣string␣literal␣at␣line␣" << line << ",␣column␣"
                    << column << endl;
                exit(1);
            }

            return src.substr(start + 1, pos - start - 2);
        }

        string consumeCharLiteral() {
            pos++;
            if (pos < src.size() && src[pos + 1] == '\'') {
                string charLiteral(1, src[pos]);
                pos += 2;
                return charLiteral;
            }
            cout << "Error:␣Invalid␣character␣literal␣on␣line␣" << line << endl;
            exit(1);
        }

        string consumeWord()
        {
            size_t start = pos;
            while(pos < src.size() && isalnum(src[pos]))
            {
                pos++;
            }
            return src.substr(start, pos - start);
        }

        vector<Token> tokenize()
        {
            vector<Token> tokens;
            while (pos < src.size())
            {
                char current = src[pos];
                if (isspace(current))
                {
                    if (current == '\n') {
                        line++;
                        column = 1;
                    } else {
                        column++;
                    }
                    pos++;
                    continue;
                }
                //comments code
                if(current=='/' && pos+1<src.size() && src[pos+1]=='/')
                {
                    while(pos<src.size() && src[pos]!='\n')
                    {
                        pos++;
                    }
                    continue;
                }
                if (isdigit(current)) {
                    tokens.push_back(Token{T_NUM, consumeNumber(), line, column});
                    continue;
                }

                if (isalpha(current))
                {
                    string word = consumeWord();
                    if (word == "int") tokens.push_back(Token{T_INT, word, line, column});
                    else if (word == "if") tokens.push_back(Token{T_IF, word, line, column});
                    else if (word == "agar") tokens.push_back(Token{T_AGAR, word, line,
                        column});
                    else if (word == "else") tokens.push_back(Token{T_ELSE, word, line,
                        column});
                    else if (word == "return") tokens.push_back(Token{T_RETURN, word, line,
                        column});
                    else if (word == "for") tokens.push_back(Token{T_FOR, word, line, column
                        });
                    else if (word == "while") tokens.push_back(Token{T_WHILE, word, line,
                        column});
                    else if (word == "float") tokens.push_back(Token{T_FLOAT, word, line,
                        column});
                    else if (word == "string") tokens.push_back(Token{T_STRING, word, line,
                        column});
                    else if (word == "char") tokens.push_back(Token{T_CHAR, word, line,
                        column});
                    else tokens.push_back(Token{T_ID, word, line, column});
                    continue;
                }
                if (current == '"') {
                    string str = consumeString();
                    tokens.push_back(Token{T_STRING, str, line, column});
                    continue;
                }

                if (current == '\'') {
                    string str = consumeCharLiteral();
                    tokens.push_back(Token{T_CHAR, str, line, column});
                    continue;
                }

                if (current == '=' && pos + 1 < src.size() && src[pos + 1] == '=')
                {
                    tokens.push_back(Token{T_EQ, "==", line, column});
                    pos += 2;
                    column += 2;
                    continue;
                }

                if (current == '<' && pos + 1 < src.size() && src[pos + 1] == '=')
                {
                    tokens.push_back(Token{T_LE, "<=", line, column});
                    pos += 2;
                    column += 2;
                    continue;
                }

                if (current == '>' && pos + 1 < src.size() && src[pos + 1] == '=')
                {
                    tokens.push_back(Token{T_GE, ">=", line, column});
                    pos += 2;
                    column += 2;
                    continue;
                }

```

```


                if (current == '&' && pos + 1 < src.size() && src[pos + 1] == '&')
                {
                    tokens.push_back(Token{T_AND, "&&", line, column});
                    pos += 2;
                    column += 2;
                    continue;
                }

                if (current == '|' && pos + 1 < src.size() && src[pos + 1] == '|')
                {
                    tokens.push_back(Token{T_OR, "||", line, column});
                    pos += 2;
                    column += 2;
                    continue;
                }

                if (current == '!' && pos + 1 < src.size() && src[pos + 1] == '=')
                {
                    tokens.push_back(Token{T_NOT, "!=", line, column});
                    pos += 2;
                    column += 2;
                    continue;
                }

                switch (current)
                {
                    case '=': tokens.push_back(Token{T_ASSIGN, "=", line, column}); break;
                    case '+': tokens.push_back(Token{T_PLUS, "+", line, column}); break;
                    case '-': tokens.push_back(Token{T_MINUS, "-", line, column}); break;
                    case '*': tokens.push_back(Token{T_MUL, "*", line, column}); break;
                    case '/': tokens.push_back(Token{T_DIV, "/", line, column}); break;
                    case '(': tokens.push_back(Token{T_LPAREN, "(", line, column}); break;
                    case ')': tokens.push_back(Token{T_RPAREN, ")", line, column}); break;
                    case '{': tokens.push_back(Token{T_LBRACE, "{", line, column}); break;
                    case '}': tokens.push_back(Token{T_RBRACE, "}", line, column}); break;
                    case ';': tokens.push_back(Token{T_SEMICOLON, ";", line, column}); break;
                    case '>': tokens.push_back(Token{T_GT, ">", line, column}); break;
                    case '<': tokens.push_back(Token{T_LT, "<", line, column}); break;
                    default: cout << "Unexpected␣character:␣" << current << "␣at␣line␣" <<
                        line << ",␣column␣" << column << endl; exit(1);
                }
                pos++;
            }
            tokens.push_back(Token{T_EOF,"", line, column});
            return tokens;
        }
};


class Parser
{
    public:
        Parser(const vector<Token> &tokens, SymbolTable &symTable, IntermediateCodeGnerator &
            icg)
        : tokens(tokens), symTable(symTable), pos(0), icg(icg)
        {
            //Constructor
            //here the private member of this class are being initalized with the arguments
                passed to this constructor
        }

        void parseProgram() {
        while (tokens[pos].type != T_EOF)
```

```
            {
                parseStatement();
            }
            cout << "Parsing␣completed␣successfully!␣No␣Syntax␣Error" << endl;

            symTable.display();
        }

    private:
        vector<Token> tokens;
        size_t pos;
        SymbolTable& symTable;
        IntermediateCodeGnerator &icg;

        void parseStatement()
        {
            if (tokens[pos].type == T_INT || tokens[pos].type == T_FLOAT || tokens[pos].type
                == T_STRING || tokens[pos].type == T_CHAR) {
                parseDeclaration();
            } else if (tokens[pos].type == T_ID) {
                parseAssignment();
            } else if (tokens[pos].type == T_IF) {
                parseIfStatement();
            } else if (tokens[pos].type == T_AGAR) {
                parseAgarStatement();
            } else if (tokens[pos].type == T_RETURN) {
                parseReturnStatement();
            } else if (tokens[pos].type == T_FOR) {
                parseForLoop();
            }
            else if (tokens[pos].type == T_WHILE) {
                parseWhileStatement();
            }
             else if (tokens[pos].type == T_LBRACE) {
                parseBlock();
            } else {
                cout << "Syntax␣error:␣unexpected␣token␣" << tokens[pos].value << "'␣at␣line␣
                    " << tokens[pos].lineNumber << endl;
                exit(1);
            }
        }

        void parseForLoop()
        {
            expect(T_FOR);           // Expect 'for'
            expect(T_LPAREN);        // Expect '('

            // Parse initialization
            expect(T_INT);
            expect(T_ID);
            expect(T_ASSIGN);
            parseExpression();       // Initialize (e.g., int i = 0)
            expect(T_SEMICOLON);     // Expect ';' after initialization

            // Parse condition
            parseExpression();       // Condition (e.g., i < 5)
            expect(T_SEMICOLON);     // Expect ';' after condition

            // Parse increment
            expect(T_ID);           // Increment variable (e.g., i)
            expect(T_ASSIGN);       // Expect '=' for increment
            parseExpression();       // Expression for increment (e.g., i + 1)
            expect(T_RPAREN);        // Expect ')' after increment

            parseStatement();        // Parse the body of the for loop
        }

        void parseWhileStatement()
        {
            expect(T_WHILE);
            expect(T_LPAREN);
            parseExpression();
            expect(T_RPAREN);
            parseStatement();
        }




        /*
        parseDeclaration handles the parsing of variable declarations.
        It expects the token type to be `T_INT` (for declaring an integer type variable),
        followed by an identifier (variable name), and a semicolon to terminate the statement
            .
        It also registers the declared variable in the symbol table with type "int".
        Example:
        int x;   // This will be parsed and the symbol table will store x with type "int".
        */
        void parseDeclaration()
        {
            if (tokens[pos].type == T_INT || tokens[pos].type == T_FLOAT || tokens[pos].type
                == T_STRING || tokens[pos].type == T_CHAR)
            {
                TokenType type = tokens[pos].type;
                string dataType = tokens[pos].value;

                pos++; // Consume the type token (T_INT, T_FLOAT, or T_STRING, or T_BOOl)

                Token idToken = tokens[pos];
                expect(T_ID); // Expect an identifier token for the variable name

                // Check for redeclaration (if variable already exists in the symbol table)
                if (symTable.lookup(idToken.value) != "")
                {
                    cout << "Semantic␣error:␣Variable␣'" << idToken.value << "'␣is␣already␣
                        declared␣at␣line␣"
                        << idToken.lineNumber << ",␣column␣" << idToken.columnNumber << endl;
                    exit(1); // Exit the program in case of semantic error
                }

                // Insert variable into the symbol table
                symTable.insert(idToken.value, dataType);

                if (tokens[pos].type == T_ASSIGN)
                {
                    pos++; // Consume the assignment token '='
                    if (tokens[pos].type == T_NUM)
                    {
                        // If assigned value is a number, check if it's a float or integer
                        string assignedValue = tokens[pos].value;
                        bool isFloat = assignedValue.find('.') != string::npos;

                        // Retrieve the variable type from the symbol table
                        string varType = symTable.lookup(idToken.value);

                        if (varType == "int" && isFloat)
                        {
                            cout << "Semantic␣error:␣Cannot␣assign␣a␣float␣value␣to␣an␣int␣
                                variable␣'"
                                << idToken.value << "'␣at␣line␣" << idToken.lineNumber
                                << ",␣column␣" << idToken.columnNumber << endl;
                            exit(1); // Exit the program in case of semantic error
                        }
                        pos++; // Consume the number token
                    }

                    else if (tokens[pos].type == T_STRING)
                    {
                        // If assigned value is a string, check if the variable is a string type
                        string varType = symTable.lookup(idToken.value);
                        if (varType != "string")
                        {
                            cout << "Semantic␣error:␣Cannot␣assign␣a␣string␣value␣to␣a␣non-string␣variable␣'"
                                << idToken.value << "'␣at␣line␣" << idToken.lineNumber
                                << ",␣column␣" << idToken.columnNumber << endl;
                            exit(1);
                        }
                        pos++; // Consume the string token
                    }

                    else if (tokens[pos].type == T_CHAR)
                    {
                        // If assigned value is a char, check if the variable is a char type
                        string varType = symTable.lookup(idToken.value);
                        if (varType != "char")
                        {
                            cout << "Semantic␣error:␣Cannot␣assign␣a␣string␣value␣to␣a␣non-string␣variable␣'"
                                << idToken.value << "'␣at␣line␣" << idToken.lineNumber
                                << ",␣column␣" << idToken.columnNumber << endl;
                            exit(1);
                        }
                        pos++; // Consume the char token
                    }

                    else
                    {
                        cout << "Syntax␣error:␣expected␣value␣after␣'='␣but␣found␣" << tokens[pos].value << endl;
                        exit(1);
                    }
                }

                expect(T_SEMICOLON); // Expect a semicolon to end the declaration
            }
            else
            {
                cout << "Syntax␣error:␣expected␣int,␣float,␣or␣string␣but␣found␣" << tokens[pos].value << endl;
                exit(1);
            }
        }


        void parseAssignment() {
            string varName = expectAndReturnValue(T_ID);
            symTable.lookup(varName);    // Ensure the variable is declared in the symbol table.


            //expect(T_ID); // Expect an identifier for the variable name



            expect(T_ASSIGN);
            string expr = parseExpression();

            icg.addInstruction(varName + " = " + expr);  // Generate intermediate code for
                the assignment.
            expect(T_SEMICOLON);
        }



        /*
            parseIfStatement handles the parsing of ‘if‘ statements.
            It expects the keyword ‘if‘, followed by an expression in parentheses that serves
                 as the condition.
            If the condition evaluates to true, it executes the statement inside the block.
                If an ‘else‘ part is present,
            it executes the corresponding statement after the ‘else‘ keyword.
            Intermediate code for the ‘if‘ statement is generated, including labels for
                conditional jumps.
            Example:
            if(5 > 3) { x = 20; }  --> This will generate intermediate code for the condition
                 check and jump instructions.
        */
        void parseIfStatement() {
            expect(T_IF);
            expect(T_LPAREN);// Expect and consume the opening parenthesis for the condition.
            string cond = parseExpression(); // Parse the condition expression inside the
                parentheses.
            expect(T_RPAREN);

            string temp = icg.newTemp();    // Generate a new temporary variable for the
                condition result.
            icg.addInstruction(temp + " = " + cond);        // Generate intermediate code for
                 storing the condition result.
            icg.addInstruction("if " + temp + " goto L1");   // Jump to label L1 if condition
                 is true.
            icg.addInstruction("goto L2");                  // Otherwise, jump to label L2.
            icg.addInstruction("L1:");                      // Otherwise, jump to label L2.

            parseStatement();

            if (tokens[pos].type == T_ELSE) {// If an ‘else‘ part exists, handle it.
                icg.addInstruction("goto L3");
                icg.addInstruction("L2:");
                expect(T_ELSE);
                parseStatement(); // Parse the statement inside the else block.
                icg.addInstruction("L3:");
            }
            else {
                icg.addInstruction("L2:");
            }
        }

        void parseAgarStatement() {
            expect(T_AGAR);
            expect(T_LPAREN);// Expect and consume the opening parenthesis for the condition.
            string cond = parseExpression(); // Parse the condition expression inside the
                parentheses.
            expect(T_RPAREN);

            string temp = icg.newTemp();    // Generate a new temporary variable for the
                condition result.
            icg.addInstruction(temp + " = " + cond);        // Generate intermediate code for
                 storing the condition result.
            icg.addInstruction("agar␣" + temp + "␣goto␣L1");   // Jump to label L1 if condition is true.
            icg.addInstruction("goto␣L2");                     // Otherwise, jump to label L2.
            icg.addInstruction("L1:");                         // Otherwise, jump to label L2.
            parseStatement();
        }

        /*
            parseReturnStatement handles the parsing of ‘return‘ statements.
            It expects the keyword ‘return‘, followed by an expression to return, and a semicolon to terminate the statement.
            It generates intermediate code to represent the return of the expression.
            Example:
            return x + 5;   -->  This will generate intermediate code like ‘return x + 5‘.
        */
        void parseReturnStatement() {
            expect(T_RETURN);
            string expr = parseExpression();
            icg.addInstruction("return␣" + expr);  // Generate intermediate code for the return statement.
            expect(T_SEMICOLON);
        }

        /*
            parseBlock handles the parsing of block statements, which are enclosed in curly braces ‘{ }‘.
            It parses the statements inside the block recursively until it reaches the closing brace.
            Example:
            { x = 10; y = 20; }   -->  This will parse each statement inside the block.
        */
        void parseBlock()
        {
            expect(T_LBRACE);  // Expect and consume the opening brace ‘{‘.
            while (tokens[pos].type != T_RBRACE && tokens[pos].type != T_EOF) {
                parseStatement();// Parse the statements inside the block.
            }
            expect(T_RBRACE);
        }

        /*
            parseExpression handles the parsing of expressions involving addition, subtraction, or comparison operations.
            It first parses a term, then processes addition (‘+‘) or subtraction (‘-‘) operators if present, generating
            intermediate code for the operations.
            Example:
            5 + 3 - 2;  -->  This will generate intermediate code like ‘t0 = 5 + 3‘ and ‘t1 = t0 - 2‘.
        */
        string parseExpression() {
            string term = parseTerm();
            while (tokens[pos].type == T_PLUS || tokens[pos].type == T_MINUS) {
                TokenType op = tokens[pos++].type;
                string nextTerm = parseTerm();    // Parse the next term in the expression.
                string temp = icg.newTemp();     // Generate a temporary variable for the result
                icg.addInstruction(temp + "␣=␣" + term + (op == T_PLUS ? "␣+␣" : "␣-␣") + nextTerm); // Intermediate code for operation
                term = temp;
            }
            if (tokens[pos].type == T_GT || tokens[pos].type == T_LT || tokens[pos].type == T_LE || tokens[pos].type == T_EQ || tokens[pos].type == T_GE) {
                pos++;
                string nextExpr = parseExpression();    // Parse the next expression for the comparison.
                string temp = icg.newTemp();              // Generate a temporary variable for the result.
                icg.addInstruction(temp + "␣=␣" + term + "␣>␣" + nextExpr); // Intermediate code for the comparison.
                term = temp;
            }
            if (tokens[pos].type == T_AND) {
                pos++;
                string nextExpr = parseExpression();  // Parse the next expression for the logical ’&&’
                string temp = icg.newTemp();          // Generate a temporary variable for the result
                icg.addInstruction(temp + "␣=␣" + term + "␣&&␣" + nextExpr); // Intermediate code for logical AND
                term = temp;  // Update term with the result of the logical AND operation
            }

            if (tokens[pos].type == T_OR) {
                pos++;
                string nextExpr = parseExpression();  // Parse the next expression for the logical ’&&’
                string temp = icg.newTemp();          // Generate a temporary variable for the result
                icg.addInstruction(temp + "␣=␣" + term + "␣||␣" + nextExpr); // Intermediate code for logical AND
                term = temp;  // Update term with the result of the logical AND operation
            }

            return term;
        }

        /*
            parseTerm handles the parsing of terms involving multiplication or division operations.
            It first parses a factor, then processes multiplication (‘*‘) or division (‘/‘) operators if present,
            generating intermediate code for the operations.
            Example:
            5 * 3 / 2;   This will generate intermediate code like ‘t0 = 5 * 3‘ and ‘t1 = t0 / 2‘.
        */
        string parseTerm() {
            string factor = parseFactor();
            while (tokens[pos].type == T_MUL || tokens[pos].type == T_DIV) {
                TokenType op = tokens[pos++].type;
                string nextFactor = parseFactor();
                string temp = icg.newTemp(); // Generate a temporary variable for the result.
                icg.addInstruction(temp + "␣=␣" + factor + (op == T_MUL ? "␣*␣" : "␣/␣") + nextFactor);  // Intermediate code for operation.
                factor = temp;  // Update the factor to be the temporary result.
            }
            return factor;
        }

        /*
            parseFactor handles the parsing of factors in expressions, which can be either numeric literals, identifiers
            (variables), or expressions inside parentheses (for sub-expressions).
            Example:
            5;          -->  This will return the number "5".
            x;          -->  This will return the identifier "x".
            (5 + 3);    --> This will return the sub-expression "5 + 3".
        */
```

```
        string parseFactor() {
            if (tokens[pos].type == T_NUM || tokens[pos].type == T_ID) {
                return tokens[pos++].value;
            } else if (tokens[pos].type == T_LPAREN) {
                expect(T_LPAREN);
                string expr = parseExpression();
                expect(T_RPAREN);
                return expr;
            } else {
                cout << "Syntax␣error:␣unexpected␣token␣'" << tokens[pos].value << "'␣at␣line␣" << tokens[pos].lineNumber << endl;
                exit(1);
            }
        }

        /*
            expect function:
            This functin is used to check whether the current token matches the expected type.
            If the token type does not match the expected type, an error message is displayed
            and the program exits. If the token type matches, it advances the position to the next token.
        */
        void expect(TokenType type) {
            if (tokens[pos].type == type) {
                pos++;
            } else {
                cout << "Syntax␣error:␣expected␣'" << type << "'␣at␣line␣" << tokens[pos].lineNumber << endl;
                exit(1);
            }
        }

        /*
        Explanation:
        - The `expect` function ensures that the parser encounters the correct tokens in the expected order.
        - It's mainly used for non-value-based tokens, such as keywords, operators, and delimiters (e.g., semicolons).
        - If the parser encounters an unexpected token, it halts the process by printing an error message, indicating where the error occurred (line number) and what was expected.
        - The `pos++` advances to the next token after confirming the expected token is present.

        Use Case:
        - This function is helpful when checking for the correct syntax or structure in a language's grammar, ensuring the parser processes the tokens in the correct order.
        */
        string expectAndReturnValue(TokenType type) {
            string value = tokens[pos].value;
            expect(type);
            return value;
        }

/*
    Why both functions are needed:
    - The `expect` function is useful when you are only concerned with ensuring the correct token type without needing its value.
    - For example, ensuring a semicolon `;` or a keyword `if` is present in the source code.
    - The `expectAndReturnValue` function is needed when the parser not only needs to check for a specific token but also needs to use the value of that token in the next stages of compilation or interpretation.
    - For example, extracting the name of a variable (`T_ID`) or the value of a constant (`T_NUMBER`) to process it in a symbol table or during expression evaluation.
*/
};

int main(int argc, char* argv[]) {
    if (argc < 2) {
        cout << "Usage: " << argv[0] << " <source file>" << endl;
        return 1;
    }

    ifstream file(argv[1]);
    if (!file.is_open()) {
        cout << "Failed to open file: " << argv[1] << endl;
        return 1;
    }

    string input((istreambuf_iterator<char>(file)), istreambuf_iterator<char>());
    file.close();




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

    for (const auto& token : tokens) {
        cout << "Token: " << token.value << ", Type: " << token.type << ", Line: " << token.lineNumber << ", Col: " << token.columnNumber << endl;
    }

    SymbolTable symTable;
    IntermediateCodeGnerator icg;
    Parser parser(tokens, symTable, icg);
    parser.parseProgram();
    cout<< endl;
    cout<< "Three Addres Code: " << endl;
    icg.printInstructions();
    icg.generateAssemblyCode();
    return 0;
}
```

Listing 7.1: C++ code snippet

7.1 Output

```
D:\7 semester\CC Lab\Lab14>compiler3.exe code.cpp
Token: int, Type: 0, Line: 1, Col: 1
Token: a, Type: 1, Line: 1, Col: 2
Token: =, Type: 6, Line: 1, Col: 3
Token: 5, Type: 2, Line: 1, Col: 4
Token: ;, Type: 15, Line: 1, Col: 4
Token: int, Type: 0, Line: 2, Col: 1
Token: b, Type: 1, Line: 2, Col: 2
Token: ;, Type: 15, Line: 2, Col: 2
Token: b, Type: 1, Line: 3, Col: 1
Token: =, Type: 6, Line: 3, Col: 2
Token: a, Type: 1, Line: 3, Col: 3
Token: +, Type: 7, Line: 3, Col: 4
Token: 10, Type: 2, Line: 3, Col: 5
Token: ;, Type: 15, Line: 3, Col: 5
Token: float, Type: 23, Line: 5, Col: 1
Token: c, Type: 1, Line: 5, Col: 2
```

```
Token: ;, Type: 15, Line: 5, Col: 2
Token: c, Type: 1, Line: 6, Col: 1
Token: =, Type: 6, Line: 6, Col: 2
Token: 2.5, Type: 2, Line: 6, Col: 3
Token: ;, Type: 15, Line: 6, Col: 3
Token: string, Type: 24, Line: 7, Col: 1
Token: name, Type: 1, Line: 7, Col: 2
Token: =, Type: 6, Line: 7, Col: 3
Token: uswa, Type: 24, Line: 7, Col: 4
Token: ;, Type: 15, Line: 7, Col: 4
Token: char, Type: 30, Line: 8, Col: 1
Token: ch, Type: 1, Line: 8, Col: 2
Token: =, Type: 6, Line: 8, Col: 3
Token: a, Type: 30, Line: 8, Col: 4
Token: ;, Type: 15, Line: 8, Col: 4
Token: agar, Type: 25, Line: 10, Col: 1
Token: (, Type: 11, Line: 10, Col: 1
Token: b, Type: 1, Line: 10, Col: 1
Token: >=, Type: 28, Line: 10, Col: 2
Token: 5, Type: 2, Line: 10, Col: 6
Token: &&, Type: 22, Line: 10, Col: 7
Token: c, Type: 1, Line: 10, Col: 10
Token: ==, Type: 20, Line: 10, Col: 11
Token: 2.5, Type: 2, Line: 10, Col: 14
Token: ), Type: 12, Line: 10, Col: 14
Token: {, Type: 13, Line: 10, Col: 15
Token: return, Type: 5, Line: 11, Col: 5
Token: b, Type: 1, Line: 11, Col: 6
Token: ;, Type: 15, Line: 11, Col: 6
Token: }, Type: 14, Line: 12, Col: 1
Token: for, Type: 18, Line: 15, Col: 1
Token: (, Type: 11, Line: 15, Col: 2
Token: int, Type: 0, Line: 15, Col: 2
Token: i, Type: 1, Line: 15, Col: 3
Token: =, Type: 6, Line: 15, Col: 4
Token: 0, Type: 2, Line: 15, Col: 5
Token: ;, Type: 15, Line: 15, Col: 5
Token: i, Type: 1, Line: 15, Col: 6
Token: <, Type: 29, Line: 15, Col: 7
Token: b, Type: 1, Line: 15, Col: 8
Token: ;, Type: 15, Line: 15, Col: 8
Token: i, Type: 1, Line: 15, Col: 9
Token: =, Type: 6, Line: 15, Col: 10
Token: i, Type: 1, Line: 15, Col: 11
Token: +, Type: 7, Line: 15, Col: 12
Token: 1, Type: 2, Line: 15, Col: 13
Token: ), Type: 12, Line: 15, Col: 13
Token: {, Type: 13, Line: 15, Col: 14
Token: if, Type: 3, Line: 16, Col: 5
Token: (, Type: 11, Line: 16, Col: 6
Token: b, Type: 1, Line: 16, Col: 6
Token: >, Type: 16, Line: 16, Col: 7
Token: 10, Type: 2, Line: 16, Col: 8
Token: ), Type: 12, Line: 16, Col: 8
Token: {, Type: 13, Line: 16, Col: 9
Token: return, Type: 5, Line: 17, Col: 9
Token: b, Type: 1, Line: 17, Col: 10
Token: ;, Type: 15, Line: 17, Col: 10
Token: }, Type: 14, Line: 18, Col: 5
Token: else, Type: 4, Line: 18, Col: 6
Token: {, Type: 13, Line: 18, Col: 7
Token: return, Type: 5, Line: 19, Col: 9
Token: 0, Type: 2, Line: 19, Col: 10
Token: ;, Type: 15, Line: 19, Col: 10
Token: }, Type: 14, Line: 20, Col: 5
Token: }, Type: 14, Line: 21, Col: 1
Token: int, Type: 0, Line: 23, Col: 1
Token: x, Type: 1, Line: 23, Col: 2
Token: ;, Type: 15, Line: 23, Col: 2
Token: x, Type: 1, Line: 24, Col: 1
Token: =, Type: 6, Line: 24, Col: 2
Token: 10, Type: 2, Line: 24, Col: 3
Token: ;, Type: 15, Line: 24, Col: 3
Token: while, Type: 19, Line: 25, Col: 1
Token: (, Type: 11, Line: 25, Col: 2
Token: x, Type: 1, Line: 25, Col: 2
Token: >, Type: 16, Line: 25, Col: 3
Token: 0, Type: 2, Line: 25, Col: 4
Token: ), Type: 12, Line: 25, Col: 4
Token: {, Type: 13, Line: 25, Col: 5
Token: x, Type: 1, Line: 26, Col: 5
Token: =, Type: 6, Line: 26, Col: 6
Token: x, Type: 1, Line: 26, Col: 7
Token: -, Type: 8, Line: 26, Col: 8
Token: 1, Type: 2, Line: 26, Col: 9
Token: ;, Type: 15, Line: 26, Col: 9
Token: }, Type: 14, Line: 27, Col: 1
Token: return, Type: 5, Line: 28, Col: 1
Token: x, Type: 1, Line: 28, Col: 2
Token: ;, Type: 15, Line: 28, Col: 2
Token: , Type: 17, Line: 28, Col: 2
Parsing completed successfully! No Syntax Error

Symbol Table:
Name: a, Type: int
Name: b, Type: int
Name: c, Type: float
Name: ch, Type: char
Name: name, Type: string
Name: x, Type: int

Three Addres Code:
Three-Address Code (TAC):
t0 = a + 10
b = t0
c = 2.5
t1 = c > 2.5
t2 = 5 && t1
t3 = b > t2
t4 = t3
agar t4 goto L1
goto L2
L1:
return b
t5 = i > b
t6 = i + 1
t7 = b > 10
t8 = t7
if t8 goto L1
goto L2
L1:
return b
goto L3
L2:
return 0
L3:
x = 10
t9 = x > 0
t10 = x - 1
x = t10
```

```
return x

Assembly Code:
LOAD a, R1
ADD 10, R1
STORE R1, t0
LOAD t0, R2
STORE R2, b
LOAD 2.5, R3
STORE R3, c
LOAD t3, R7
STORE R7, t4
JMP L2
L1:
LOAD i, R13
ADD 1, R13
STORE R13, t6
LOAD t7, R15
STORE R15, t8
JMP L2
L1:
JMP L3
L2:
L3:
LOAD 10, R24
STORE R24, x
LOAD x, R26
SUB 1, R26
STORE R26, t10
LOAD t10, R27
STORE R27, x
```

Listing 7.2: C++ code snippet

Chapter 8

Conclusion

In this report, we explored the key stages of a compiler, focusing on code generation and optimization. Each phase plays a vital role in transforming high-level source code into efficient machine code. We emphasized the importance of accurate syntax analysis, semantic checks, and the generation of intermediate representations, such as Three-Address Code (TAC), as the backbone of further optimizations. The assembly code generation phase demonstrated how high-level operations are translated into machine-level instructions, with attention to register allocation, memory management, and control flow handling.

The key takeaway from this compiler design is the significance of performance and correctness. Optimizing register usage and minimizing memory access were critical objectives, ensuring that the generated code runs efficiently on the target machine. Moreover, the careful handling of control flow operations, such as conditional and unconditional jumps, ensures that the program logic is preserved during translation.

In conclusion, this compiler report highlights the complexity and precision required in modern compilers to generate optimized and correct assembly code. Through the successful implementation of these phases, we have shown how abstract programming constructs can be systematically translated into executable machine code, achieving both correctness and efficiency. Further enhancements, such as advanced optimizations and support for additional language features, can be explored to improve the compiler's performance and extend its capabilities.