Compiler Construction: Final Documentation

Session: 2021 – 2025

Submitted by:

M. Jawad Haider 2021-CS-149

Supervised by:

Sir Laeeq Khan

Department of Computer Science

University of Engineering and Technology Lahore

Pakistan

Contents

1 Introduction 1
2 Objectives 1
3 Scope 2
4 Features 3
4.1 Variable Declarations 3
4.2 Arithmetic Operations 3
4.3 Conditional Statements (`if, else`) 3
4.4 Loops - `for` and `while` 3
4.5 Return Statements 3
5 Architecture and Modules 4
5.1 Lexer 4
5.2 Parser 4
5.3 Intermediate Code Generator (ICG) 4
5.4 Assembly Code Generator 5
5.5 Utilities 5
6 Implementation Details and Code 5
6.1 Lexical Analysis 5
6.2 Parsing 11
6.3 Intermediate Code Generation 23
6.4 Assembly Code Generation 25
7 Input and Output 31
7.1 Input 31
7.2 Output 32
8 Future Improvements 32
9 Conclusion 32

1 Introduction

A compiler is a program that converts high-level source code into machine-executable instructions. The compilation process involves multiple phases such as lexical analysis, syntax analysis, semantic analysis, and code generation.

This project focuses on building a custom compiler for a simple high-level language supporting basic programming constructs like:

- **Variable declarations**
- **Arithmetic operations**
- **Conditional statements `(if, else)`**
- **Loops `(for, while)`**
- **Return statements**

The compiler generates:

1. **Three-Address Code (TAC)**: An intermediate representation used for code optimization and analysis.
2. **x86 Assembly Code**: Low-level instructions that can be executed on an x86-based system.

2 Objectives

The main objectives of this project are:

1. **To understand the phases of compilation**: Implementing lexical analysis, parsing, and code generation.
2. **To develop a working compiler** that converts high-level code into optimized x86 assembly instructions.
3. **To generate intermediate code** (Three-Address Code) as a bridge between parsing and machine-level code generation.
4. **To create a modular and extensible design**: Each phase (lexer, parser, code generation) is implemented as a separate module.

5. **To support key programming constructs** like variables, arithmetic operations, loops, and conditionals.
6. **To provide meaningful output** in the form of Three-Address and Assembly Code.

3 Scope

The scope of this project includes:

1. **Basic Programming Constructs:**
   - Variable declarations
   - Assignments
   - Arithmetic expressions
   - Conditional statements `(if-else)`
   - Loops `(for and while)`
   - Return statements
2. **Compilation Phases:**
   - Lexical Analysis
   - Syntax Analysis
   - Intermediate Code Generation (TAC)
   - Assembly Code Generation for x86 architecture
3. **Supported Data Types:**
   - `int` (integer)
   - `float` (floating-point number)
   - `string` (character string)
4. **Outputs:**
   - Intermediate Three-Address Code (TAC)
   - x86 Assembly Code

The project is limited to handling simple programs without advanced features like functions, arrays, or object-oriented constructs.

4 Features

4.1 Variable Declarations

Example:

```
int a;
float b = 2.5;
string name = "compiler";
```

4.2 Arithmetic Operations

Addition, subtraction, multiplication, and division. Example:

```
int sum = a + b * 2;
```

4.3 Conditional Statements (`if, else`)

Example:

```
if (a > b) {
    return 1;
} else {
    return 0;
}
```

4.4 Loops - `for` and `while`

Example:

```
for (int i = 0; i < 10; i++) {
    sum = sum + i;
}

while (x > 0) {
    x = x - 1;
}
```

4.5 Return Statements

Example:

```
    return sum;
```

5 Architecture and Modules

The project is divided into five main modules:

5.1 Lexer

- **Purpose**: Breaks the input code into a series of tokens.
- **File**: `lexer.cpp`
- **Key Responsibilities**:
  - Tokenize keywords, identifiers, numbers, strings, and operators.
  - Skip comments and whitespace.

5.2 Parser

- **Purpose**: Ensures the program follows valid syntax and generates intermediate code.
- **File**: `parser.cpp`
- **Key Responsibilities**:
  - Parses tokens produced by the lexer.
  - Validates syntax for constructs like declarations, conditionals, loops, and assignments.
  - Invokes the Intermediate Code Generator for TAC generation.

5.3 Intermediate Code Generator (ICG)

- **Purpose**: Generates Three-Address Code (TAC) as an intermediate representation.
- **File**: `intermediateCodeGenerator.cpp`
- **Output Example**:

```
temp_0 = a * 2
temp_1 = b + temp_0
if temp_1 goto L1
```

```
goto L2
L1:
return temp_1
```

5.4 Assembly Code Generator

- **Purpose**: Converts TAC into x86 assembly instructions.
- **File**: `assemblyGenerator.cpp`
- **Output Example**:

```
MOV EAX, a
IMUL EAX, 2
MOV EBX, b
ADD EAX, EBX
CMP EAX, 0
JNE L1
L1:
MOV EAX, EAX
int 0x80
```

5.5 Utilities

- **Purpose**: Contains helper functions for token management and comparison operator checks.
- **File**: `utils.cpp`

6 Implementation Details and Code

The compiler operates in the following phases:

6.1 Lexical Analysis

Breaks input into tokens.

```
class Lexer
{
```

```
private:
    string src;
    size_t position;
    size_t lineNumber;
    /*
    It hold positive values.
    In C++, size_t is an unsigned integer data type used to
   represent the
    size of objects in bytes or indices, especially when
   working with memory-related
    functions, arrays, and containers like vector or string.
   You can also use the int data type but size_t is recommended
    one
    */

public:
    Lexer(const string &src)
    {
        this->src = src;
        this->position = 0;
        this->lineNumber = 0;
    }

    vector<Token> tokenize()
    {
        vector<Token> tokens;
        while (position < src.size())
        {
            char current = src[position];

            if (current == '\n')
            {
                lineNumber++;
                position++;
                continue;
            }

            // Detect Single line comments
            if (current == '/' && src[position + 1] == '/')
            {
```

```
            while (src[position] != '\n' && position <= src
.size())
            {
                position++;
            }
            continue;
        }

        // Read string values
        if (current == '"')
        {
            position++;
            // string str = "\"";
            string str = "";
            while (src[position] != '"')
            {
                str = str + src[position];
                position++;
            }
            // str += src[position];
            position++;
            tokens.push_back(Token{T_STRING, str,
lineNumber});
            continue;
        }

        if (isspace(current))
        {
            position++;
            continue;
        }
        if (isdigit(current))
        {
            tokens.push_back(Token{T_NUM, consumeNumber(),
lineNumber});
            continue;
        }
        if (isalpha(current))
        {
            string word = consumeWord();
```

```
                if (word == "int")
                    tokens.push_back(Token{T_INT, word,
lineNumber});
                else if (word == "float")
                    tokens.push_back(Token{T_FLOAT, word,
lineNumber});
                else if (word == "string")
                    tokens.push_back(Token{T_STRING, word,
lineNumber});
                else if (word == "if")
                    tokens.push_back(Token{T_IF, word,
lineNumber});
                else if (word == "else")
                    tokens.push_back(Token{T_ELSE, word,
lineNumber});
                else if (word == "return")
                    tokens.push_back(Token{T_RETURN, word,
lineNumber});
                else if (word == "while")
                    tokens.push_back(Token{T_WHILE, word,
lineNumber});
                else if (word == "for")
                    tokens.push_back(Token{T_FOR, word,
lineNumber});
                else
                    tokens.push_back(Token{T_ID, word,
lineNumber});
                continue;
            }

            // Handle Multi-character Operators (==, <=, >=)
            if (current == '=' && src[position + 1] == '=')
            {
                tokens.push_back(Token{T_EQ, "==", lineNumber})
;
                position += 2;
                continue;
            }
            else if (current == '!' && src[position + 1] == '='
)
```

```
            {
                tokens.push_back(Token{T_NEQ, "!=", lineNumber
});
                position += 2;
                continue;
            }
            else if (current == '<' && src[position + 1] == '='
)
            {
                tokens.push_back(Token{T_LE, "<=", lineNumber})
;
                position += 2;
                continue;
            }
            else if (current == '>' && src[position + 1] == '='
)
            {
                tokens.push_back(Token{T_GE, ">=", lineNumber})
;
                position += 2;
                continue;
            }

            // Add OPERATORS in the tokens vector
            switch (current)
            {
            case '=':
                tokens.push_back(Token{T_ASSIGN, "=",
lineNumber});
                break;
            case '+':
                tokens.push_back(Token{T_PLUS, "+", lineNumber
});
                break;
            case '-':
                tokens.push_back(Token{T_MINUS, "-", lineNumber
});
                break;
            case '*':
```

```
                tokens.push_back(Token{T_MUL, "*", lineNumber})
;
                break;
            case '/':
                tokens.push_back(Token{T_DIV, "/", lineNumber})
;
                break;
            case '(':
                tokens.push_back(Token{T_LPAREN, "(",
lineNumber});
                break;
            case ')':
                tokens.push_back(Token{T_RPAREN, ")",
lineNumber});
                break;
            case '{':
                tokens.push_back(Token{T_LBRACE, "{",
lineNumber});
                break;
            case '}':
                tokens.push_back(Token{T_RBRACE, "}",
lineNumber});
                break;
            case ';':
                tokens.push_back(Token{T_SEMICOLON, ";",
lineNumber});
                break;
            case '>':
                tokens.push_back(Token{T_GT, ">", lineNumber});
                break;
            case '<':
                tokens.push_back(Token{T_LT, "<", lineNumber});
                break;
            default:
                cout << "Unexpected character: " << current <<
endl;
                exit(1);
            }
            position++;
        }
```

```
        tokens.push_back(Token{T_EOF, "", lineNumber});

        // printTokens(tokens);
        return tokens;
    }

    string consumeNumber()
    {
        size_t start = position;
        while (position < src.size() && isdigit(src[position]))
            position++;
        return src.substr(start, position - start);
    }

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

    void printTokens(vector<Token> tokens)
    {
        for (size_t i = 0; i < tokens.size(); i++)
        {
            cout << tokens[i].value << "\t" << getTokenName(
   tokens[i].type) << "\t" << tokens[i].lineNumber << endl;
        }
    }
};
```

6.2 Parsing

Generates the syntax tree and validates constructs.

```
class Parser
{

public:
```

```
    Parser(const vector<Token> &tokens, SymbolTable &
   symbolTable, IntermediateCodeGenerator &icg)
        : tokens(tokens), position(0), symbolTable(symbolTable)
   , icg(icg)
    {
        this->dataTypes[T_INT] = T_INT;
        this->dataTypes[T_STRING] = T_STRING;
        this->dataTypes[T_FLOAT] = T_FLOAT;
        this->dataTypes[T_CHAR] = T_CHAR;

        this->blockStatement[T_IF] = T_IF;
        this->blockStatement[T_WHILE] = T_WHILE;
        this->blockStatement[T_FOR] = T_FOR;
    }

    void parseProgram()
    {
        while (tokens[position].type != T_EOF)
        {
            // cout << "before: " << tokens[position].value <<
   endl;
            parseStatement();
            // cout << "before: " << tokens[position].value <<
   endl;
        }

        symbolTable.displaySymbolTable();
    }

private:
    vector<Token> tokens;
    size_t position;
    map<TokenType, TokenType> dataTypes;
    map<TokenType, TokenType> blockStatement;
    SymbolTable &symbolTable;
    IntermediateCodeGenerator &icg;

    void parseStatement()
    {
```

```
    // cout << "tokens[position].value: " << tokens[
position].value << endl;
    if (dataTypes.find(tokens[position].type) != dataTypes.
end())
    {
        parseDeclaration(dataTypes[tokens[position].type]);
    }
    else if (tokens[position].type == T_ID)
    {
        parseAssignment();
    }
    else if (blockStatement.find(tokens[position].type) !=
blockStatement.end())
    {
        parseBlockStatement(tokens[position].type);
    }
    else if (tokens[position].type == T_RETURN)
    {
        parseReturnStatement();
    }
    else if (tokens[position].type == T_LBRACE)
    {
        parseBlock();
    }
    else
    {
        cout << "Syntax error: unexpected token " <<
getQuotesAroundStr(tokens[position].value) << endl;
        exit(1);
    }
}

void parseBlock()
{
    expect(T_LBRACE);
    while (tokens[position].type != T_RBRACE && tokens[
position].type != T_EOF)
    {
        parseStatement();
    }
```

```
        expect(T_RBRACE);
    }

    void parseDeclaration(TokenType dataType)
    {
        expect(dataType);
        string identifierName = tokens[position].value;
        expect(T_ID);
        Token symbolInstance;
        if (tokens[position].type == T_ASSIGN)
        {
            expect(T_ASSIGN);
            if (tokens[position].type == T_STRING)
            {
                symbolInstance = Token{T_STRING, tokens[
position].value};
                expect(T_STRING);
            }
            else if (tokens[position].type == T_NUM || tokens[
position].type == T_ID)
            {
                symbolInstance = parseAndEvaluateExpression();
            }
        }
        symbolInstance.type = dataType;
        symbolTable.declareVariable(identifierName,
symbolInstance);
        expect(T_SEMICOLON);
        if (symbolInstance.icgVariable != "")
            icg.addInstruction(identifierName + " = " +
symbolInstance.icgVariable);
        else if (symbolInstance.value != "")
        {
            if (symbolInstance.type == T_STRING)
            {
                symbolInstance.value = "\"" + symbolInstance.
value + "\"";
            }
            icg.addInstruction(identifierName + " = " +
symbolInstance.value);
```

```
        }
    }

    string parseAssignment(bool generateIntermediateCode = true
)
    {
        string symbol = tokens[position].value;
        expect(T_ID);
        Token symbolInstance = symbolTable.getVariableToken(
symbol);
        if (tokens[position].type == T_PLUS || tokens[position
].type == T_MINUS)
        {
            parseIncrementDecrementOperator(symbol, &
symbolInstance);
        }
        else
        {
            expect(T_ASSIGN);
            symbolInstance = parseAndEvaluateExpression(
symbolInstance);
            expect(T_SEMICOLON);
        }
        symbolTable.updateVariable(symbol, symbolInstance);
        string assignmentTo = symbolInstance.icgVariable == ""
? symbolInstance.value : symbolInstance.icgVariable;
        string icgInstruction = symbol + " = " + assignmentTo;
        if (generateIntermediateCode)
        {
            icg.addInstruction(icgInstruction); // Generate
intermediate code for the assignment.
        }
        return icgInstruction;
    }

    void parseBlockStatement(TokenType blockStatementKeyword)
    {
        expect(blockStatementKeyword);
        expect(T_LPAREN);
        string loopStartLabel;
```

```
        if (blockStatementKeyword == T_FOR)
        {
            // Initialization / Declaration of iterator
            if (dataTypes.find(tokens[position].type) !=
    dataTypes.end())
            {
                parseDeclaration(tokens[position].type);
            }
            else if (tokens[position].type == T_ID)
            {
                parseAssignment();
            }
        }

        if (blockStatementKeyword == T_WHILE ||
    blockStatementKeyword == T_FOR)
        {
            // Starting label of FOR loop
            loopStartLabel = icg.newLabel();
            icg.addInstruction(loopStartLabel + ":");
        }

        // Evaluating condition
        Token condition = parseAndEvaluateExpression();

        string trueConditionLabel = icg.newLabel();
        string falseConditionLabel = icg.newLabel();
        icg.addInstruction("if " + condition.icgVariable + "
    goto " + trueConditionLabel);
        icg.addInstruction("goto " + falseConditionLabel);
        icg.addInstruction(trueConditionLabel + ":");

        string iteratorInstruction; // To receive instruction
    for iterator part of FOR loop
        if (blockStatementKeyword == T_FOR)
        {
            expect(T_SEMICOLON);
            iteratorInstruction = parseAssignment(false);
        }
```

```
        expect(T_RPAREN);

        parseStatement(); // Body of IF/FOR/WHILE

        if (blockStatementKeyword == T_FOR)
        {
            icg.addInstruction(iteratorInstruction); //
    Iterator instruction before going to start of loop
            icg.addInstruction("goto " + loopStartLabel);
            icg.addInstruction(falseConditionLabel + ":");
        }
        else if (blockStatementKeyword == T_WHILE)
        {
            icg.addInstruction("goto " + loopStartLabel);
            icg.addInstruction(falseConditionLabel + ":");
        }
        else if (blockStatementKeyword == T_IF && tokens[
    position].type == T_ELSE)
        {
            string elseLabel = icg.newLabel();
            icg.addInstruction("goto " + elseLabel);
            icg.addInstruction(falseConditionLabel + ":");
            expect(T_ELSE);
            parseStatement();
            icg.addInstruction(elseLabel + ":");
        }
    }

    void parseReturnStatement()
    {
        expect(T_RETURN);
        Token exp = parseAndEvaluateExpression();
        expect(T_SEMICOLON);
        string statement = exp.icgVariable == "" ? exp.value :
    exp.icgVariable;
        icg.addInstruction("return " + statement);
    }

    void parseIncrementDecrementOperator(string identifier,
    Token *identifierValue)
```

```
{
    if (tokens[position].type == T_PLUS)
    {
        expect(T_PLUS);
        expect(T_PLUS);
        int value = stoi(identifierValue->value);
        identifierValue->value = to_string(value + 1);
        identifierValue->icgVariable = identifier + " + 1";
    }
    else if (tokens[position].type == T_MINUS)
    {
        expect(T_MINUS);
        expect(T_MINUS);
        int value = stoi(identifierValue->value);
        identifierValue->value = to_string(value - 1);
        identifierValue->icgVariable = identifier + " - 1";
    }
}

Token parseAndEvaluateExpression(Token initialValue = {})
{
    Token result = initialValue;
    Token firstTerm = parseTerm();
    result.icgVariable = firstTerm.icgVariable == "" ?
firstTerm.value : firstTerm.icgVariable;
    if (firstTerm.type == T_ID)
    {
        Token symbolInstance = symbolTable.getVariableToken
(firstTerm.value);
        string identifierValue = symbolInstance.value;
        TokenType identifierType = symbolInstance.type;

        if (identifierValue == "")
            showErrorMessagesAndExit(getQuotesAroundStr(
firstTerm.value) + " has value undefined!");

        result.value = identifierValue;
    }
    else
    {
```

```
            result.value = firstTerm.value;
        }
        while (tokens[position].type == T_PLUS || tokens[
position].type == T_MINUS)
        {
            string op = tokens[position].type == T_PLUS ? "+" :
 "-";
            position++;
            bool isNextTermIdentifier = false;
            Token nextTerm = parseTerm(&isNextTermIdentifier);
            string nextTermValue = nextTerm.value;
            TokenType nextTermType = result.type;
            if (nextTerm.type == T_ID)
            {
                Token tempSymbolInstance = symbolTable.
getVariableToken(nextTerm.value);
                nextTermValue = tempSymbolInstance.value;
                nextTermType = tempSymbolInstance.type;
            }
            if (nextTermType != result.type) // Validation
check for operation between different data types
                showErrorMessagesAndExit(
                    "Operation '" + op + "' cannot be applied
between type: " + getTokenName(result.type) + " and " +
getTokenName(nextTermType) + "!");
            if (op == "+")
            {
                if (result.type == T_INT)
                    result.value = to_string(stoi(result.value)
 + stoi(nextTermValue));
                else
                    result.value += nextTermValue;
            }
            else
            {
                if (result.type == T_INT)
                    result.value = to_string(stoi(result.value)
 - stoi(nextTermValue));

                else if (result.type == T_STRING)
```

```
                    showErrorMessagesAndExit("Cannot perform
'-' op on type string");
         }
         string newVar = icg.newTemp();
         string nextTermVar = nextTerm.icgVariable == "" ?
nextTerm.value : nextTerm.icgVariable;
         cout << "isNextTermIdentifier: " <<
isNextTermIdentifier << endl;
         if (result.type == T_STRING && !
isNextTermIdentifier)
         {
             nextTermVar = "\"" + nextTermVar + "\"";
         }
         icg.addInstruction(newVar + " = " + result.
icgVariable + " " + op + " " + nextTermVar);
         result.icgVariable = newVar;
     }
     // if (tokens[position].type == T_GT || tokens[position
].type == T_LT || tokens[position].type == T_EQ)
     if (isComparisonOperator(tokens[position].type))
     {
         TokenType comparisonOp = tokens[position].type;
         position++;
         Token nextExp = parseAndEvaluateExpression();
         string nextExpVar = nextExp.icgVariable == "" ?
nextExp.value : nextExp.icgVariable;
         string icgVar = icg.newTemp();
         icg.addInstruction(icgVar + " = " + result.
icgVariable + " " + getTokenName(comparisonOp) + " " +
nextExpVar);
         result.icgVariable = icgVar;
     }
     return result;
 }

 Token parseTerm(bool *isNextTermIdentifier = nullptr)
 {
     Token factor = parseFactor();
     Token result = factor;
     if (factor.type == T_ID)
```

```
        {
            result = symbolTable.getVariableToken(factor.value)
;
            result.icgVariable = factor.value;
            if (isNextTermIdentifier != nullptr)
                *isNextTermIdentifier = true;
        }
        while (tokens[position].type == T_MUL || tokens[
position].type == T_DIV)
        {
            // position++;
            // factor = parseFactor();

            TokenType op = tokens[position++].type;
            Token nextFactor = parseFactor();
            string nextFactorValue = nextFactor.value;
            if (nextFactor.type == T_ID)
            {
                Token symbolInstance = symbolTable.
getVariableToken(nextFactor.value);

                if (symbolInstance.value == "")
                    showErrorMessagesAndExit(getQuotesAroundStr
(nextFactor.value) + " has value undefined!");

                nextFactorValue = symbolInstance.value;
                if (isNextTermIdentifier != nullptr)
                    *isNextTermIdentifier = true;
            }

            string resultStr = result.icgVariable == "" ?
result.value : result.icgVariable;
            if (op == T_MUL)
            {
                result.value = to_string(stoi(result.value) *
stoi(nextFactorValue));
            }
            else if (op == T_DIV)
            {
```

```
            result.value = to_string(stoi(result.value) /
stoi(nextFactorValue));
        }

        string temp = icg.newTemp();
        icg.addInstruction(temp + " = " + resultStr + (op
== T_MUL ? " * " : " / ") + nextFactor.value);
        result.icgVariable = temp;
    }
    return result;
}

Token parseFactor()
{
    if (tokens[position].type == T_NUM || tokens[position].
type == T_ID || tokens[position].type == T_STRING)
    {
        position++;
        return tokens[position - 1];
    }
    else if (tokens[position].type == T_LPAREN)
    {
        expect(T_LPAREN);
        Token exp = parseAndEvaluateExpression();
        expect(T_RPAREN);
        return exp;
    }
    else
    {
        cout << "Syntax error: unexpected token " <<
getQuotesAroundStr(tokens[position].value) << endl;
        exit(1);
    }
    return Token{};
}

void expect(TokenType type)
{
    if (tokens[position].type == type)
    {
```

```
            position++;
        }
        else
        {
            showErrorMessagesAndExit(
                "Syntax error: expected " + getTokenName(type)
 + " but found " + tokens[position].value,
                "Error at line number: " + to_string(tokens[
position].lineNumber));
            exit(1);
        }
    }

    template <typename... Args>
    void showErrorMessagesAndExit(const string &str, const Args
    &...args)
    {
        cout << "ERROR => " << str << endl;
        showErrorMessagesAndExit(args...);
    }

    void showErrorMessagesAndExit(const string &str)
    {
        cout << "ERROR => " << str << endl;
        exit(1);
    }

    string getQuotesAroundStr(string text)
    {
        return "'" + text + "'";
    }
};
```

6.3 Intermediate Code Generation

Produces TAC instructions.

```
class IntermediateCodeGenerator
{
public:
```

```
vector<string> instructions;
int tempCount = 0;
int labelCount = 1;

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

string newLabel()
{
    return "L" + to_string(labelCount++);
}

void addInstruction(const string &instr)
{
    if (instr[0] == 'L')
        instructions.push_back(instr);
    else
        instructions.push_back("    " + instr);
}

void writeToOutputFile(string fileName)
{
    ofstream outputFile(fileName);
    if (!outputFile.is_open())
    {
        cerr << "Error: Could not write to file " <<
fileName << endl;
        exit(1);
    }
    for (const auto &instr : instructions)
    {
        outputFile << instr << endl;
    }
    outputFile.close();
    cout << "Intermediate code written to " << "output/TAC-
Output.txt" << endl;
}
```

```
    void printInstructions()
    {
        bool isBlockStarted = false;
        for (const auto &instr : instructions)
        {
            if (instr.find("L") != std::string::npos)
            {
                cout << instr << endl;
                isBlockStarted = true;
            }
            else if (instr.find("goto") != std::string::npos)
            {
                cout << instr << endl;
                isBlockStarted = false;
            }
            else
            {
                string output = isBlockStarted ? "    " + instr
    : instr;
                cout << output << endl;
            }
        }
    }
};
```

6.4 Assembly Code Generation

Converts TAC into x86 Assembly Code.

```
class AssemblyGenerator
{
private:
    vector<string> assemblyCode;             // Holds the
   generated assembly code
    map<string, string> variableToRegister; // Maps variables
   to registers
    map<string, string> registerToVariable; // Maps registers
   to variables
    vector<string> availableRegisters;      // Pool of
   available x86 registers
```

```
    set<string> tempVariables;                    // Tracks temporary
    variables

public:
    AssemblyGenerator()
    {
        // Initialize available x86 registers
        availableRegisters = {"EAX", "EBX", "ECX", "EDX"};
    }

    // Generate x86 assembly code from TAC
    // Generate x86 assembly code from TAC
    void generateAssembly(const vector<string> &tacLines, const
    string &outputFile)
    {
        for (const string &line : tacLines)
        {
            string trimmedLine = trim(line);
            vector<string> tokens = split(trimmedLine, ' ');

            // cout << "------------------ Start
   ------------------------" << endl;
            // for (size_t i = 0; i < tokens.size(); i++)
            // {
            //     cout << "tokens[i]: " << tokens[i] << endl;
            // }
            // cout << "------------------ End
   ------------------------" << endl;


            if (tokens.empty()) continue; // Skip empty lines

            // Handle different TAC instructions
            if (tokens.size() == 3 && tokens[1] == "=") {
                handleAssignment(tokens);
            }
            else if (tokens.size() == 5 && (tokens[3] == "+" ||
    tokens[3] == "-" || tokens[3] == "*" || tokens[3] == "/"))
   {
                handleArithmetic(tokens);
```

```
        }
        else if (tokens.size() == 4 && tokens[0] == "if") {
            handleConditionalJump(tokens);
        }
        else if (tokens.size() == 2 && tokens[0] == "goto")
 {
            handleUnconditionalJump(tokens);
        }
        else if (tokens.size() == 1 && tokens[0].back() ==
':') {
            handleLabel(tokens);
        }
        else if (tokens.size() == 2 && tokens[0] == "return
") {
            handleReturn(tokens);
        }
        else if (tokens.size() == 5 && (tokens[3] == ">" ||
 tokens[3] == "<")) {
            handleComparison(tokens);
        }
        else {
            cerr << "Error: Unrecognized TAC instruction: "
 << line << endl;
        }
    }
    // Write all assembly instructions to the file
    writeToFile(outputFile);
}

// Handle simple assignments: a = b
void handleAssignment(const vector<string> &tokens)
{
    string dest = tokens[0];
    string src = tokens[2];
    assemblyCode.push_back("    MOV " + getRegister(dest) +
", " + src);
}

// Handle arithmetic operations: temp = a + b, a - b, etc.
void handleArithmetic(const vector<string> &tokens)
```

```
{
    string dest = tokens[0];
    string left = tokens[2];
    string right = tokens[4];
    string op = tokens[3];

    string leftReg = getRegister(left);

    // Load left operand into the register
    assemblyCode.push_back("    MOV " + leftReg + ", " +
left);

    if (op == "+") {
        assemblyCode.push_back("    ADD " + leftReg + ", "
+ right);
    } else if (op == "-") {
        assemblyCode.push_back("    SUB " + leftReg + ", "
+ right);
    } else if (op == "*") {
        assemblyCode.push_back("    IMUL " + leftReg + ", "
 + right);
    } else if (op == "/") {
        assemblyCode.push_back("    IDIV " + right);
    }

    // Store the result back to the destination
    assemblyCode.push_back("    MOV " + dest + ", " +
leftReg);
}

// Handle conditional jumps: if temp goto L1
void handleConditionalJump(const vector<string> &tokens)
{
    string condition = tokens[1];
    string label = tokens[3];
    assemblyCode.push_back("    CMP " + condition + ", 0");
    assemblyCode.push_back("    JNE " + label);
}

// Handle unconditional jumps: goto L1
```

```
void handleUnconditionalJump(const vector<string> &tokens)
{
    string label = tokens[1];
    assemblyCode.push_back("    JMP " + label);
}

// Handle labels: L1:
void handleLabel(const vector<string> &tokens)
{
    assemblyCode.push_back(tokens[0]);
}

// Handle return statements: return value
void handleReturn(const vector<string> &tokens)
{
    string value = tokens[1];
    assemblyCode.push_back("    MOV eax, " + value);
    assemblyCode.push_back("    int 0x80"); // Exit syscall
}

// Handle comparisons: temp = a > b or temp = a < b
void handleComparison(const vector<string> &tokens)
{
    string dest = tokens[0];
    string left = tokens[2];
    string right = tokens[4];
    string op = tokens[3];

    string leftReg = getRegister(left);

    assemblyCode.push_back("    MOV " + leftReg + ", " +
left);
    assemblyCode.push_back("    CMP " + leftReg + ", " +
right);

    if (op == ">") {
        assemblyCode.push_back("    SETg AL");
    } else if (op == "<") {
        assemblyCode.push_back("    SETl AL");
    }
```

```
    assemblyCode.push_back("    MOVzx " + dest + ", AL");
}

// Helper function to get a register for a variable
string getRegister(const string &var)
{
    if (variableToRegister.find(var) == variableToRegister.
end()) {
        string reg = availableRegisters.back();
        availableRegisters.pop_back();
        variableToRegister[var] = reg;
    }
    return variableToRegister[var];
}

// Write the generated assembly code to a file
void writeToFile(const string &outputFile)
{
    ofstream asmFile(outputFile);
    for (const auto &line : assemblyCode) {
        asmFile << line << endl;
    }
    asmFile.close();
    cout << "Assembly code generated in " << outputFile <<
endl;
}

// Helper function: Split a string by a delimiter
vector<string> split(const string &line, char delimiter)
{
    vector<string> tokens;
    string token;
    istringstream tokenStream(line);
    cout << "line.find('\"'): " << line.find('\"') << endl;
    if (line.find('\"') != string::npos) {

    }
    while (getline(tokenStream, token, delimiter)) {
        cout << "token: " << token << endl;
```

```
            if (!token.empty()) tokens.push_back(token);
        }
        return tokens;
    }

    // Helper function: Trim whitespace from a string
    string trim(const string &str)
    {
        size_t start = str.find_first_not_of(" \t");
        size_t end = str.find_last_not_of(" \t");
        return (start == string::npos || end == string::npos) ?
    "" : str.substr(start, end - start + 1);
    }

    void printAssembly()
    {
        for (const auto &line : assemblyCode)
        {
            cout << line << endl;
        }
    }
};
```

7 Input and Output

7.1 Input

The input file must contain source code written in the supported high-level language.

- File Extension: `.jwd`

Example Input:

```
    int a = 10;
    int b = 20;
    if (a > b) {
        return a;
    } else {
        return b;
    }
```

7.2 Output

1. **TAC (Three-Address Code)**:
   File: `output/TAC-Output.txt`

2. **x86 Assembly Code**:
   File: `output/Assembly-Output.txt`

8 Future Improvements

1. **Add Support for Functions**: Allow defining and calling functions.

2. **Arrays and Data Structures**: Include advanced data types like arrays and structs.

3. **Optimized Code Generation**: Implement register allocation and optimization techniques.

4. **Improved Error Handling**: Report errors with more clarity, including suggestions for fixes.

5. **Additional Control Flow Constructs**: Add support for switch-case and do-while loops.

9 Conclusion

This project successfully implements a basic compiler that translates high-level source code into optimized Three-Address Code (TAC) and x86 Assembly Code. By following a modular approach, the project demonstrates the core concepts of Compiler Construction, including lexical analysis, parsing, and code generation.

This project serves as a foundation for further improvements and can be extended to include additional programming language features and optimization techniques.