University of Engineering and Technology Lahore

Compiler Construction Project Report

Submitted to:
Mr. Laeeq Khan Niazi

Submitted by:
Samia Liaqat 2021-CS-128

December 15, 2024

Certificate

This is to certify that **Samia Liaqat** (Registration No: **2021-CS-128**) has successfully completed the Project-Based Learning (PBL) task on the topic titled **"Compiler Construction Using C++ Language"**. The project was carried out under my guidance and supervision as part of the course requirements for Compiler Construction.

The dedication and effort demonstrated by Samia Liaqat throughout the project are commendable, and the work meets the standards expected for this academic requirement.

Project Mentor	**Date:**
(Mr. Laeeq Khan Niazi)	________________

Acknowledgment

The successful completion of this project would not have been possible without the support and guidance of several individuals, to whom I owe my heartfelt gratitude.

Firstly, I would like to express my profound appreciation to my supervisor, **Mr. Laeeq Khan Niazi**, for his continuous guidance, constructive criticism, and encouragement throughout the course of this project. His invaluable insights and expertise provided a solid foundation for my work, helping me navigate challenges effectively.

I am equally indebted to my family for their unwavering support and understanding during the completion of this project. Their patience and encouragement have been a constant source of motivation.

I would also like to thank my friends and colleagues for their valuable suggestions and moral support, which significantly contributed to this project. Their collaboration and feedback have been incredibly helpful in refining my ideas and approach.

Lastly, I extend my gratitude to the institution and department for providing the necessary resources and a conducive environment for learning and development. This project has been an enriching experience, and I am grateful for the opportunity to apply theoretical knowledge to practical implementation.

Contents

1 Introduction **6**

2 Language Description **7**
2.1 Language . . . 7
2.1.1 Identifier Rules . . . 7
2.1.2 Data Types . . . 7
2.1.3 Expressions . . . 8
2.1.4 Preprocessor Directives . . . 8
2.1.5 Statements . . . 8
2.1.6 Examples of Conditional and Loop Statements . . . 9

3 Phases of Compiler **11**
3.1 Description . . . 11
3.2 Lexical Analysis . . . 11
3.3 Syntax Analysis . . . 11
3.4 Semantic Analysis . . . 11
3.5 Intermediate Code Generation . . . 12
3.6 Code Optimization . . . 12
3.7 Code Generation . . . 12
3.8 Symbol Table Management . . . 12
3.9 Assembly Code Generation . . . 12

4 Lexical Analyzer **13**
4.1 Description . . . 13
4.2 How to Implement in C++ . . . 13
4.3 Outcomes . . . 13
4.4 Functionalities Added . . . 14
4.4.1 Token Types . . . 14
4.4.2 Helper Functions . . . 15
4.4.3 End of File (EOF) Handling . . . 16
4.5 Code . . . 16

5 Syntax Analyzer **17**
5.1 Description . . . 17
5.2 How to Implement in C++ . . . 17
5.3 Outcomes . . . 17
5.4 Functionalities Added . . . 17
5.4.1 Single-Line Declaration and Initialization Handling . . . 17

5.4.2 Parsing `void` Functions . . . 18
5.4.3 Parsing `break` Statements . . . 18
5.4.4 Parsing `if` Statements . . . 18
5.4.5 Parsing `switch` Statements . . . 18
5.4.6 Parsing `return` Statements . . . 18
5.4.7 Parsing Blocks of Code . . . 18
5.4.8 Parsing `while` Loops . . . 18
5.4.9 Parsing `for` Loops . . . 18
5.4.10 Parsing `do-while` Loops . . . 19
5.4.11 Parsing Input Statements . . . 19
5.4.12 Parsing Output Statements . . . 19
5.4.13 Parsing `agar` Statements . . . 19
5.5 Code . . . 19

6 Semantic Analyzer 20
6.1 Description . . . 20
6.2 How to Implement in C++ . . . 20
6.3 Outcomes . . . 20
6.4 Functionalities Added . . . 20
6.4.1 Variable Declaration . . . 21
6.4.2 Variable Type Retrieval . . . 21
6.4.3 Check if Variable is Declared . . . 21
6.4.4 Print Symbol Table . . . 22
6.5 Code . . . 23

7 Intermediate Code Generation 24
7.1 Description . . . 24
7.2 How to Implement in C++ . . . 24
7.3 Outcomes . . . 24
7.4 Functionalities Added . . . 24
7.4.1 Generate New Temporary Variable . . . 25
7.4.2 Add Instruction . . . 25
7.4.3 Print Instructions . . . 25
7.4.4 Save Instructions to File . . . 26
7.5 Code . . . 27

8 Assembly Code Generation 28
8.1 Description . . . 28
8.2 How to Implement in C++ . . . 28
8.3 Outcomes . . . 28
8.4 Functionalities Added . . . 28
8.4.1 Main Functions . . . 28
8.4.2 Key Concepts and Instruction Types . . . 29
8.5 Code . . . 30

9 Output Screenshots of Each Phase 31
9.1 Test File . . . 31
9.2 Phase 1: Lexical Analyzer . . . 32
9.3 Phase 3: Semantic Analyzer(Symbol Table) . . . 33

9.4 Phase 4: Intermediate Code Generation 34
9.5 Phase 5: Assembly Code Generation 35

10 Feasibility and Future Scope **36**

11 Conclusion **37**

Chapter 1

Introduction

A compiler is a program that translates code written in one programming language into another language, while keeping the meaning of the original program intact. The main goal of a compiler is to produce target code that is efficient in terms of time and memory usage.

The process of compilation is divided into several steps, each of which builds upon the previous one. Each phase takes the output of the previous phase as its input, processes it, and then passes it on to the next phase. The first step is the analysis phase, where the compiler examines the source code, breaks it into smaller parts, and checks for errors such as incorrect words (lexical errors), wrong sentence structure (grammatical errors), and invalid syntax (syntax errors).

This analysis phase creates an intermediate code, which is also called assembly language code. This intermediate code is designed to be independent of any specific machine, making it easier to optimize and convert into assembly code in the later stages of compilation.

The compiler described in this report focuses on taking C++ source code as input and transforming it into intermediate code. This transformation represents the "front end" of the compiler's design, where the code is analyzed and prepared for further processing.

The intermediate code generated during this process plays a vital role in ensuring that the compiler can work across different types of machines. The subsequent stages of compilation focus on further optimizing this intermediate representation and converting it into machine-specific assembly code, completing the "back end" of the compiler process.

This report provides an in-depth exploration of the various phases of compiler construction, including their implementation in C++, the results achieved, and the additional functionalities introduced.

Chapter 2

Language Description

2.1 Language

This chapter provides the details of the language specifications.

2.1.1 Identifier Rules

The following rules define valid identifiers in the language:

1. Identifiers can be of any length.
2. Identifiers are case-sensitive.
3. Identifiers may only contain alphanumeric characters (a-z, A-Z, 0-9) and the underscore (_).
4. Reserved keywords cannot be used as identifiers.
5. Special characters such as semicolons, periods, whitespaces, slashes, or commas are not allowed within or as part of identifiers.
6. Special characters are treated as separate tokens and are not permitted inside identifiers.

2.1.2 Data Types

The language supports the following data types:

- Integer
- Float
- Double
- String
- Character
- Boolean

2.1.3 Expressions

The language supports the following types of expressions:

1. Arithmetic operations: `+`, `-`, `*`, `/`
2. Use of parentheses for grouping operations
3. Various types of numbers, including integers, floats, doubles, and scientific notation
4. String manipulation
5. Relational expressions: `>`, `<`, `>=`, `<=`, `==`, `!=`

2.1.4 Preprocessor Directives

The language supports preprocessor directives and comments:

- Single-line comments: `// Example of a single-line comment`
- Multi-line comments: `/* Example of a multi-line comment */`

2.1.5 Statements

The language allows the following types of statements:

1. Declaration statements: `int a;`
2. Assignment statements: `a = 6;`
3. Conditional statements, including:
   - Simple `if` statements
   - Nested `if` statements
   - Custom conditional statements using custom syntax
4. Iterative statements:
   - `for` loops, including nested `for` loops
   - `while` loops, including nested `while` loops
5. Increment and decrement operations: `a++` and `a--`
6. Return statements
7. Block statements

2.1.6 Examples of Conditional and Loop Statements

- **If Statement:**

```
if (x > y) {
    // Do nothing
} else {
    // Do something
}
```

- **Nested If Statement:**

```
if (x > y) {
    if (x > sum) {
        // Do nothing
    } else {
        // Do something
    }
}
```

- **While Loop:**

```
while (sum > 5) {
    x = 30;
}
```

- **For Loop:**

```
for (int i = 0; i < 5; i++) {
    // Some Code
}
```

- **Nested For Loop:**

```
for (int i = 0; i < 5; i++) {
    for (int j = i; j < 7; j++) {
        // Do something
    }
}
```

- **Nested While Loop:**

```
while (x > y) {
    while (y > sum) {
        // Do something
    }
}
```

This chapter outlines the foundational rules and capabilities of the language, providing a structured overview for implementation.

Chapter 3

Phases of Compiler

3.1 Description

The compiler's analysis phase dissects the source program into its core components and establishes a grammatical framework for them. This framework is then utilized to generate an intermediate representation of the original program. This stage is often referred to as the compiler's front end.

3.2 Lexical Analysis

The lexical analysis phase, also known as the scanner, reads the source code and breaks it into tokens. Tokens are the smallest units of a program, such as keywords, identifiers, and symbols.

- Input: Source code
- Output: Tokens

3.3 Syntax Analysis

The syntax analysis phase, or parser, checks the grammatical structure of the token sequence against the language's grammar. It generates a parse tree or abstract syntax tree (AST).

- Input: Tokens
- Output: Parse Tree or AST

3.4 Semantic Analysis

In this phase, the compiler ensures that the program's semantics are correct. It checks for type mismatches, undeclared variables, and other semantic errors.

- Input: Parse Tree or AST
- Output: Annotated Parse Tree

3.5 Intermediate Code Generation

The intermediate code generation phase converts the annotated parse tree into an intermediate representation (IR), which is a platform-independent code.

- Input: Annotated Parse Tree
- Output: Intermediate Representation (IR)

3.6 Code Optimization

This phase improves the intermediate code by optimizing it for better performance or reduced resource usage. It eliminates redundant code and simplifies expressions.

- Input: Intermediate Representation
- Output: Optimized Intermediate Representation

3.7 Code Generation

The code generation phase translates the optimized intermediate representation into target machine code. It produces the final executable code.

- Input: Optimized Intermediate Representation
- Output: Machine Code

3.8 Symbol Table Management

Throughout the compilation process, a symbol table is maintained. It stores information about variables, functions, and other entities for quick lookup.

3.9 Assembly Code Generation

The Assembly Code Generation phase translates the optimized intermediate code into low-level assembly language instructions. These instructions are specific to the target machine's architecture and act as an intermediate step between intermediate code and machine code. This phase ensures that the generated assembly code is correct, efficient, and ready for further processing by the assembler.

- **Input:** Optimized Intermediate Code
- **Output:** Assembly Language Instructions
- **Key Tasks:**
  - Register allocation and management.
  - Translation of intermediate operations to assembly instructions.
  - Handling of function calls, memory access, and control flow.

Chapter 4

Lexical Analyzer

4.1 Description

The **Lexical Analysis** phase is the first step in the compilation process. The source code is read character by character, and meaningful sequences of characters (called *lexemes*) are grouped into tokens. These tokens represent the smallest units of syntax, such as:

- Keywords (`int`, `return`)
- Operators (`+`, `-`, `=`)
- Identifiers (`x`, `main`)
- Literals (`10`, `3.14`)
- Punctuation (`;`, `(`, `)`)

Errors in this phase include unrecognized characters or malformed tokens.

4.2 How to Implement in C++

1. Use regular expressions to define patterns for tokens.
2. Implement a finite state automaton to recognize token boundaries.
3. Scan the source code line by line or character by character.

4.3 Outcomes

- A structured list of tokens.
- Errors for invalid tokens, such as unrecognized symbols.

4.4 Functionalities Added

The lexer in the provided code performs several key functionalities that are crucial for lexical analysis. Each functionality is responsible for recognizing specific patterns or tokens in the source code. Below are the details of each functionality implemented in the lexer:

4.4.1 Token Types

The lexer recognizes the following categories of tokens:

Data Types

- `T_INT`: Represents integer data types.
- `T_FLOAT`: Represents floating-point data types.
- `T_DOUBLE`: Represents double-precision floating-point data types.
- `T_STRING`: Represents string data types.
- `T_BOOL`: Represents boolean data types.
- `T_CHAR`: Represents character data types.

Identifiers and Literals

- `T_ID`: Represents variable or function names.
- `T_NUM`: Represents numeric constants.

Control Flow

- `T_IF`, `T_ELSE`: Represent conditional statements.
- `T_RETURN`: Represents the return statement.
- `T_AGAR`, `T_MAGAR`: Alternative representations for `T_IF` and `T_ELSE`.
- `T_WHILE`, `T_FOR`: Represent loops.
- `T_SWITCH`: Represents a switch-case statement.

Logical and Relational Operators

- `T_GT`: Greater than.
- `T_LT`: Less than.
- `T_EQ`: Equal to.
- `T_NE`: Not equal to.
- `T_LE`: Less than or equal to.

- `T_GE`: Greater than or equal to.
- `T_LOGICAL_AND`, `T_LOGICAL_OR`: Logical AND and OR operators.

Arithmetic Operators

- `T_PLUS`: Addition operator.
- `T_MINUS`: Subtraction operator.
- `T_MUL`: Multiplication operator.
- `T_DIV`: Division operator.

Punctuation

- `T_LPAREN`, `T_RPAREN`: Left and right parentheses.
- `T_LBRACE`, `T_RBRACE`: Left and right braces.
- `T_SEMICOLON`: Semicolon for statement termination.
- `T_COLON`: Colon used in switch-case statements.

Stream Operators and I/O Tokens

- `T_STREAM_INSERTION_OPERATOR`: Represents the `<<` operator.
- `T_STANDARD_OUTPUT_STREAM`: Represents `cout`.
- `T_EXTRACTION_OPERATOR`: Represents the `>>` operator.
- `T_STARNDARD_INPUT_STREAM`: Represents `cin`.

Other Tokens

- `T_PREPROCESSOR`: Represents preprocessor directives starting with `#`.
- `T_TRUE`, `T_FALSE`: Boolean values.
- `T_UNKNOWN`: Represents unknown or invalid tokens.
- `T_EOF`: End-of-file token.

4.4.2 Helper Functions

The lexical analyzer includes various helper functions to process and generate tokens:

`consumeNumber`

Processes numeric literals and determines whether they are integers or floating-point numbers using the `isFloat` flag.

`consumeWord`

Processes keywords and identifiers by reading sequences of alphabetic characters.

`consumeString`

Processes string literals enclosed in quotation marks.

`skipComments`

Skips over single-line and multi-line comments to avoid tokenizing them.

`tokenTypeToString`

Converts a token type into its string representation for debugging and display purposes.

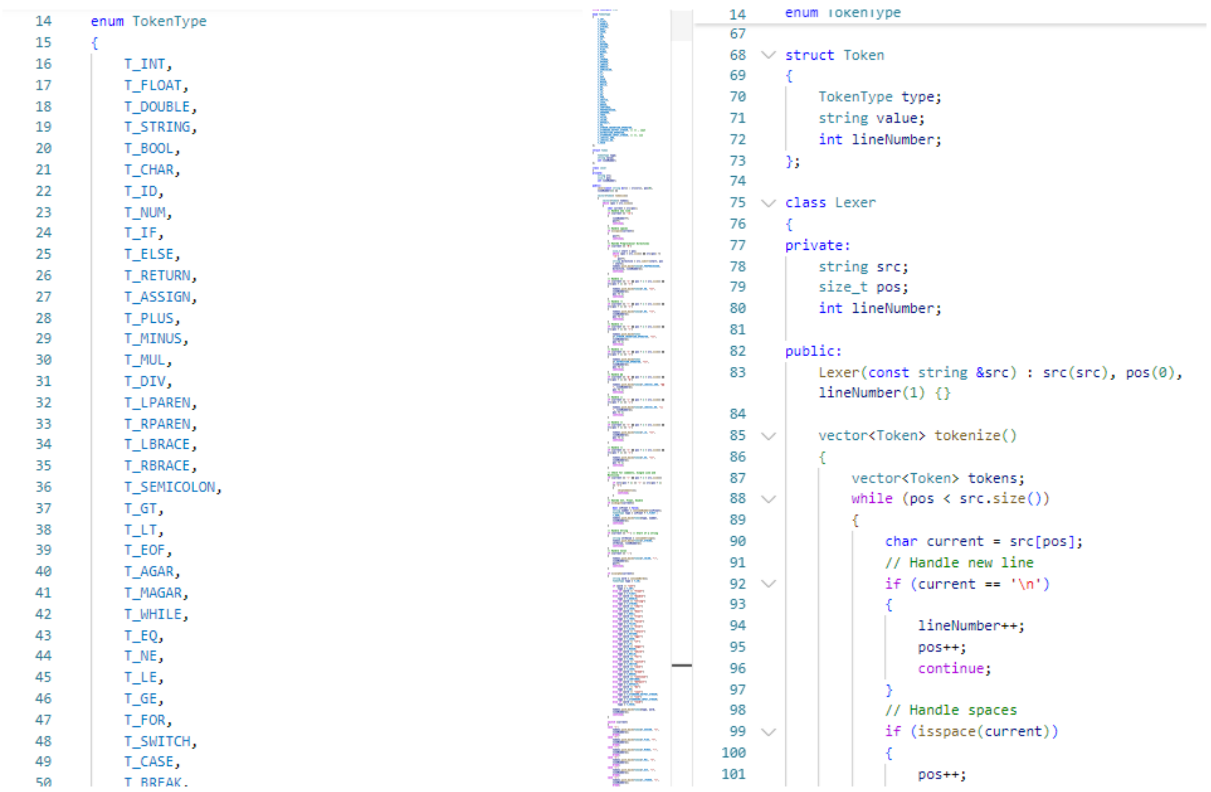
`printTokenizer`

Prints all tokens generated by the lexer for debugging and verification purposes.

4.4.3 End of File (EOF) Handling

Finally, the lexer adds a special `T_EOF` token to indicate the end of the source code. This marks the end of the tokenization process and signifies that the lexer has processed all the input.

4.5 Code



```cpp
enum TokenType
{
    T_INT,
    T_FLOAT,
    T_DOUBLE,
    T_STRING,
    T_BOOL,
    T_CHAR,
    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_LT,
    T_EOF,
    T_AGAR,
    T_MAGAR,
    T_WHILE,
    T_EQ,
    T_NE,
    T_LE,
    T_GE,
    T_FOR,
    T_SWITCH,
    T_CASE,
    T_BREAK,
```

```cpp
enum TokenType

struct Token
{
    TokenType type;
    string value;
    int lineNumber;
};

class Lexer
{
private:
    string src;
    size_t pos;
    int lineNumber;

public:
    Lexer(const string &src) : src(src), pos(0),
    lineNumber(1) {}

    vector<Token> tokenize()
    {
        vector<Token> tokens;
        while (pos < src.size())
        {
            char current = src[pos];
            // Handle new line
            if (current == '\n')
            {
                lineNumber++;
                pos++;
                continue;
            }
            // Handle spaces
            if (isspace(current))
            {
                pos++;
```

Chapter 5

Syntax Analyzer

5.1 Description

The **Syntax Analysis** phase checks whether the sequence of tokens conforms to the grammar rules of the programming language. It constructs a **parse tree** or an **Abstract Syntax Tree (AST)** to represent the hierarchical structure of the source code.

5.2 How to Implement in C++

1. Define grammar rules, such as:

```
<assignment> ::= <identifier> '=' <expression> ';'
```

2. Implement a parser using:
   - Recursive descent parsing (top-down parsing).
   - Shift-reduce parsing (bottom-up parsing).

5.3 Outcomes

- A structured representation of the source code in the form of an AST.
- Errors for misplaced tokens, missing semicolons, etc.

5.4 Functionalities Added

5.4.1 Single-Line Declaration and Initialization Handling

The compiler efficiently processes single-line declarations and assignments. The `parseDeclarationOr DeclarationAssignment()` function identifies whether a token corresponds to a variable declaration or a combined declaration and initialization, ensuring proper syntax validation and intermediate code generation.

5.4.2 Parsing `void` Functions

Functions with a `void` return type are parsed and processed using the `parseVoidFunction()` function. This functionality ensures that `void` functions are correctly identified and their syntax is validated. The parser handles the function definition and ensures compatibility with the overall program structure.

5.4.3 Parsing `break` Statements

The compiler supports the parsing and handling of `break` statements using the `parseBreakStatement()` function. These statements are typically used to exit loops or `switch` cases, and the system ensures they are correctly placed and translated into appropriate intermediate and target code.

5.4.4 Parsing `if` Statements

Conditional constructs, such as `if` statements, are processed by the `parseIfStatement()` function. This functionality validates the condition's syntax and generates the necessary branching logic for execution.

5.4.5 Parsing `switch` Statements

The `parseSwitchStatement()` function enables the compiler to process `switch` statements. It handles the parsing of `case` labels, the `default` label, and ensures the generation of proper branching logic.

5.4.6 Parsing `return` Statements

The `parseReturnStatement()` function processes `return` statements. These statements are used to return control from a function to the calling function, and the compiler ensures proper syntax validation and code generation.

5.4.7 Parsing Blocks of Code

Using the `parseBlock()` function, the compiler processes blocks of code enclosed in braces ({}). This ensures that scoped code is parsed and analyzed correctly for both syntax and semantics.

5.4.8 Parsing `while` Loops

`while` loops are parsed using the `parseWhileStatement()` function. The lexer identifies the `while` keyword, and the parser validates the loop condition and body, generating the necessary intermediate code.

5.4.9 Parsing `for` Loops

The `parseForStatement()` function handles the parsing of `for` loops. It processes initialization, condition, and iteration expressions to ensure proper validation and intermediate code generation.

5.4.10 Parsing `do-while` Loops

The `parseDoWhileStatement()` function supports the handling of `do-while` loops. The parser validates the loop body and ensures the loop condition is correctly checked at the end of each iteration.

5.4.11 Parsing Input Statements

The `parseInputStatement()` function processes standard input stream statements (`cin`) in the source code. It validates syntax involving the extraction operator (`>>`) and generates intermediate code for runtime execution.

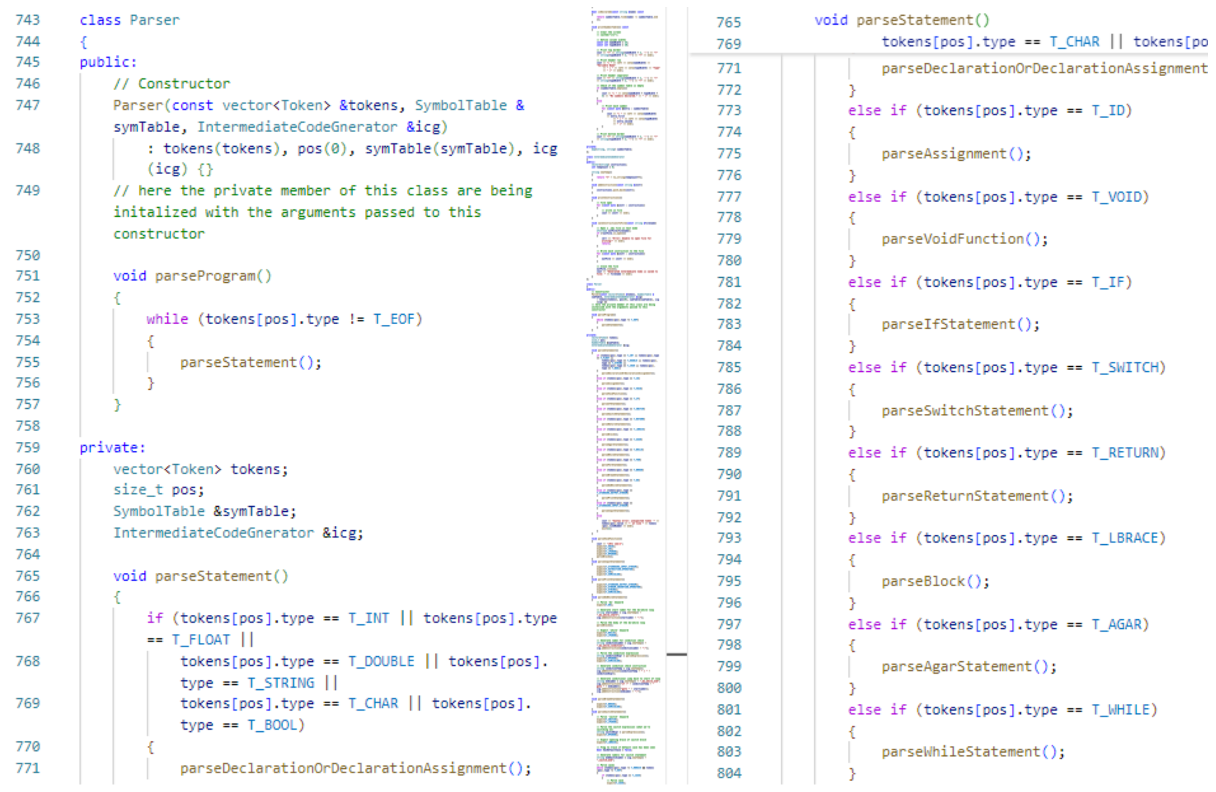
5.4.12 Parsing Output Statements

The `parsePrintStatement()` function handles standard output stream statements (`cout`). The lexer identifies the `cout` keyword and the insertion operator (`<<`), while the parser ensures syntax validation and generates intermediate code.

5.4.13 Parsing `agar` Statements

The `parseAgarStatement()` function processes `agar` statements, a non-standard construct. The parser validates its syntax and ensures proper integration with the program logic.

5.5 Code



```
class Parser
{
public:
    // Constructor
    Parser(const vector<Token> &tokens, SymbolTable &symTable, IntermediateCodeGenerator &icg)
        : tokens(tokens), pos(0), symTable(symTable), icg(icg) {}
    // 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();
        }
    }

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

    void parseStatement()
    {
        if (tokens[pos].type == T_INT || tokens[pos].type == T_FLOAT ||
            tokens[pos].type == T_DOUBLE || tokens[pos].type == T_STRING ||
            tokens[pos].type == T_CHAR || tokens[pos].type == T_BOOL)
        {
            parseDeclarationOrDeclarationAssignment();
```

```
    void parseStatement()
            tokens[pos].type == T_CHAR || tokens[po
            parseDeclarationOrDeclarationAssignment
        }
        else if (tokens[pos].type == T_ID)
        {
            parseAssignment();
        }
        else if (tokens[pos].type == T_VOID)
        {
            parseVoidFunction();
        }
        else if (tokens[pos].type == T_IF)
        {
            parseIfStatement();
        }
        else if (tokens[pos].type == T_SWITCH)
        {
            parseSwitchStatement();
        }
        else if (tokens[pos].type == T_RETURN)
        {
            parseReturnStatement();
        }
        else if (tokens[pos].type == T_LBRACE)
        {
            parseBlock();
        }
        else if (tokens[pos].type == T_AGAR)
        {
            parseAgarStatement();
        }
        else if (tokens[pos].type == T_WHILE)
        {
            parseWhileStatement();
        }
```

Chapter 6

Semantic Analyzer

6.1 Description

The **Semantic Analysis** phase ensures that the code makes sense semantically:

- Type checking: Ensures type compatibility (e.g., assigning an `int` to a `float`).
- Variable scope: Checks if variables are declared before use.
- Function calls: Validates argument types and counts.

6.2 How to Implement in C++

1. Use a symbol table to track variable names, types, and scope.
2. Validate each statement for semantic correctness.

6.3 Outcomes

- Detects errors like:
  - Using undeclared variables.
  - Assigning incompatible types.
  - Invalid function calls.
- Outputs a semantically valid intermediate representation.

6.4 Functionalities Added

The `SymbolTable` class is a key component in managing the variables within a compiler or interpreter. It ensures that variables are declared before use, allows for type retrieval, and maintains the integrity of the program's semantic structure. The class is built to handle variable declarations, type checking, and symbol management throughout the program's analysis phase.

6.4.1 Variable Declaration

Functionality: The `declareVariable` function is used to declare a new variable and add it to the symbol table. Each variable has a name and a type, which are both stored in the symbol table. Before adding the variable, the function checks if a variable with the same name already exists in the symbol table. If the variable is already declared, a runtime error is thrown, ensuring that duplicate declarations are prevented.

Function Prototype:

```
void declareVariable(const string &name, const string &type);
```

Description: This function checks the symbol table to ensure that the variable is not already declared. If the variable already exists, it throws an error, indicating a semantic mistake in the program. Otherwise, it adds the variable to the symbol table with its corresponding type. This helps in maintaining the uniqueness of variables in the scope of the program.

Usage Example:

```
symbolTable.declareVariable("x", "int"); % Declares 'x' as an integer
```

The variable `x` is successfully declared as an integer if it doesn't already exist in the symbol table. Otherwise, an error message will be triggered.

6.4.2 Variable Type Retrieval

Functionality: The `getVariableType` function retrieves the type of a variable given its name. It checks whether the variable is present in the symbol table and returns the type associated with the variable. If the variable is not declared, an error is thrown, indicating that the requested variable doesn't exist.

Function Prototype:

```
string getVariableType(const string &name);
```

Description: This function is used during semantic analysis when the compiler or interpreter needs to verify the type of a variable before performing operations on it. It ensures that any variable used in an expression has been declared and its type is correctly assigned. If the variable has not been declared, an error is raised.

Usage Example:

```
string type = symbolTable.getVariableType("x"); % Retrieves the type of 'x'
```

In this example, the type of the variable `x` is fetched from the symbol table. If `x` is declared, its type (e.g., ”int”) will be returned. If not, a runtime error will occur.

6.4.3 Check if Variable is Declared

Functionality: The `isDeclared` function checks if a variable has already been declared in the symbol table. It returns `true` if the variable exists in the symbol table, and `false` otherwise. This function is useful for preventing the use of undeclared variables in the program.

Function Prototype:

```
bool isDeclared(const string &name) const;
```

Description: This function is used to determine if a variable has been previously declared, which is important for both syntactic and semantic checks. If the variable exists in the symbol table, it returns `true`, otherwise `false`. This can be used to avoid errors such as using an undeclared variable, which would otherwise cause compilation or runtime failures.

Usage Example:

```
bool exists = symbolTable.isDeclared("x"); % Checks if 'x' is declared
```

Here, the function checks if `x` exists in the symbol table. It returns a boolean value indicating whether the variable `x` is declared or not.

6.4.4 Print Symbol Table

Functionality: The `printSymbolTable` function displays the contents of the symbol table, showing all the declared variables and their types. The symbol table is printed in a structured tabular format, making it easy to read and verify the variables declared within the program. If no variables are declared, it prints a message indicating that the symbol table is empty.

Function Prototype:

```
void printSymbolTable() const;
```

Description: This function provides a visual representation of the symbol table, listing all the variables that have been declared along with their respective types. It helps in debugging and verifying that the program's variables are being correctly managed. If the table is empty (no variables are declared), it outputs a message stating "No symbols declared." The function uses a well-defined table format to make it easier to read the data.

Usage Example:

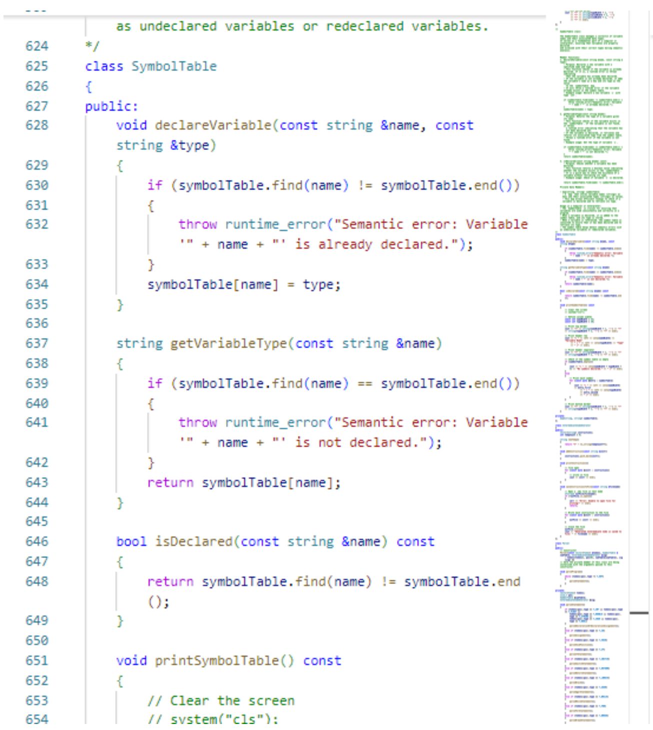
```
symbolTable.printSymbolTable(); % Prints the entire symbol table
```

The output will display a neatly formatted table with the names and types of all declared variables. If no variables are declared, it will print a message such as:

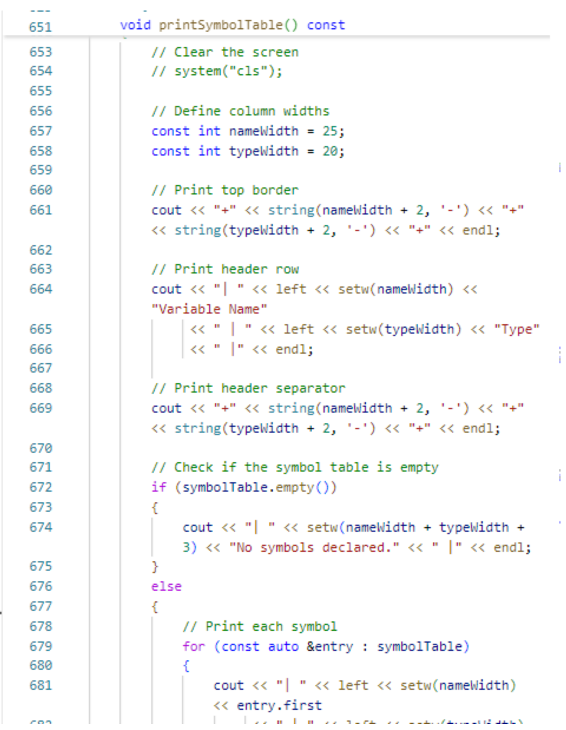
```
+-------------------------+----------------------+
| Variable Name           | Type                 |
+-------------------------+----------------------+
| x                       | int                  |
+-------------------------+----------------------+
```

In this case, `x` is declared as an integer. The table layout makes it easy to visually inspect the list of variables and their types.

6.5 Code



```
    as undeclared variables or redeclared variables.
*/
class SymbolTable
{
public:
    void declareVariable(const string &name, const string &type)
    {
        if (symbolTable.find(name) != symbolTable.end())
        {
            throw runtime_error("Semantic error: Variable '" + name + "' is already declared.");
        }
        symbolTable[name] = type;
    }

    string getVariableType(const string &name)
    {
        if (symbolTable.find(name) == symbolTable.end())
        {
            throw runtime_error("Semantic error: Variable '" + name + "' is not declared.");
        }
        return symbolTable[name];
    }

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

    void printSymbolTable() const
    {
        // Clear the screen
        // system("cls");
```



```
    void printSymbolTable() const

        // Clear the screen
        // system("cls");

        // Define column widths
        const int nameWidth = 25;
        const int typeWidth = 20;

        // Print top border
        cout << "+" << string(nameWidth + 2, '-') << "+" << string(typeWidth + 2, '-') << "+" << endl;

        // Print header row
        cout << "| " << left << setw(nameWidth) << "Variable Name"
             << " | " << left << setw(typeWidth) << "Type"
             << " |" << endl;

        // Print header separator
        cout << "+" << string(nameWidth + 2, '-') << "+" << string(typeWidth + 2, '-') << "+" << endl;

        // Check if the symbol table is empty
        if (symbolTable.empty())
        {
            cout << "| " << setw(nameWidth + typeWidth + 3) << "No symbols declared." << " |" << endl;
        }
        else
        {
            // Print each symbol
            for (const auto &entry : symbolTable)
            {
                cout << "| " << left << setw(nameWidth) << entry.first
```

Chapter 7

Intermediate Code Generation

7.1 Description

The **Intermediate Code Generation** phase translates the source program into an intermediate representation (IR). This IR is a simplified and platform-independent version of the source code, often in the form of three-address code or abstract syntax trees.

7.2 How to Implement in C++

1. Traverse the Abstract Syntax Tree (AST) created during the syntax analysis phase.
2. Generate platform-independent code, such as:

```
t1 = a + b
t2 = t1 * c
t3 = t2 / d
```

7.3 Outcomes

- An intermediate representation of the source code.
- Errors related to invalid expressions or undefined behavior.

7.4 Functionalities Added

The `IntermediateCodeGenerator` class is a crucial component in a compiler that generates intermediate code from source code. This intermediate code serves as a bridge between the high-level source code and the machine code. The class includes functions for creating temporary variables, adding instructions, printing intermediate code to the console, and saving the generated code to a file.

7.4.1 Generate New Temporary Variable

Functionality: The `newTemp` function is responsible for generating new temporary variables with a unique identifier. These temporary variables are used to store intermediate values during the compilation process. Each time `newTemp` is called, it generates a new temporary variable (e.g., t0, t1, t2, etc.) by appending an incremented counter value to the base name "t".

Function Prototype:

```
string newTemp();
```

Description: The function keeps track of the number of temporary variables created through the `tempCount` variable, which is incremented every time a new temporary variable is generated. The function returns the name of the temporary variable as a string.

Usage Example:

```
string tempVar = codeGen.newTemp();
% Generates a new temporary variable like 't0'
```

In this example, calling `newTemp` generates a new temporary variable (e.g., `t0`) and returns it as a string.

7.4.2 Add Instruction

Functionality: The `addInstruction` function adds a new instruction to the list of intermediate code instructions. This is used to generate the intermediate representation of the program during the compilation process.

Function Prototype:

```
void addInstruction(const string &instr);
```

Description: The function takes an instruction (in string format) and appends it to the `instructions` vector. This helps in building a sequence of intermediate code that represents the program's behavior.

Usage Example:

```
codeGen.addInstruction("t0 = x + y");
% Adds the instruction to the list of intermediate code
```

Here, the instruction `"t0 = x + y"` is added to the intermediate code list. Each call to `addInstruction` appends a new instruction to the vector, creating the intermediate code representation of the program.

7.4.3 Print Instructions

Functionality: The `printInstructions` function is used to print the list of intermediate code instructions to the console. It iterates over all instructions stored in the `instructions` vector and outputs them line by line.

Function Prototype:

```
void printInstructions();
```

Description: This function allows the user to view the generated intermediate code directly in the console for debugging or inspection purposes. It is helpful during the compilation process to ensure that the intermediate code is being generated correctly.

Usage Example:

```
codeGen.printInstructions();
% Prints all the generated instructions to the console
```

When invoked, this function outputs all the intermediate code instructions line by line in the console.

7.4.4 Save Instructions to File

Functionality: The `saveInstructionsToFile` function saves the intermediate code instructions to a file. This is particularly useful for storing the intermediate code in a file format that can be further processed by subsequent stages of the compiler or for later analysis.

Function Prototype:

```
void saveInstructionsToFile(const string &filename);
```

Description: The function opens a file in text mode using the provided filename and writes all the instructions stored in the `instructions` vector to the file. After writing, the file is closed, ensuring the intermediate code is saved properly. If the file cannot be opened, an error message is displayed.

Usage Example:

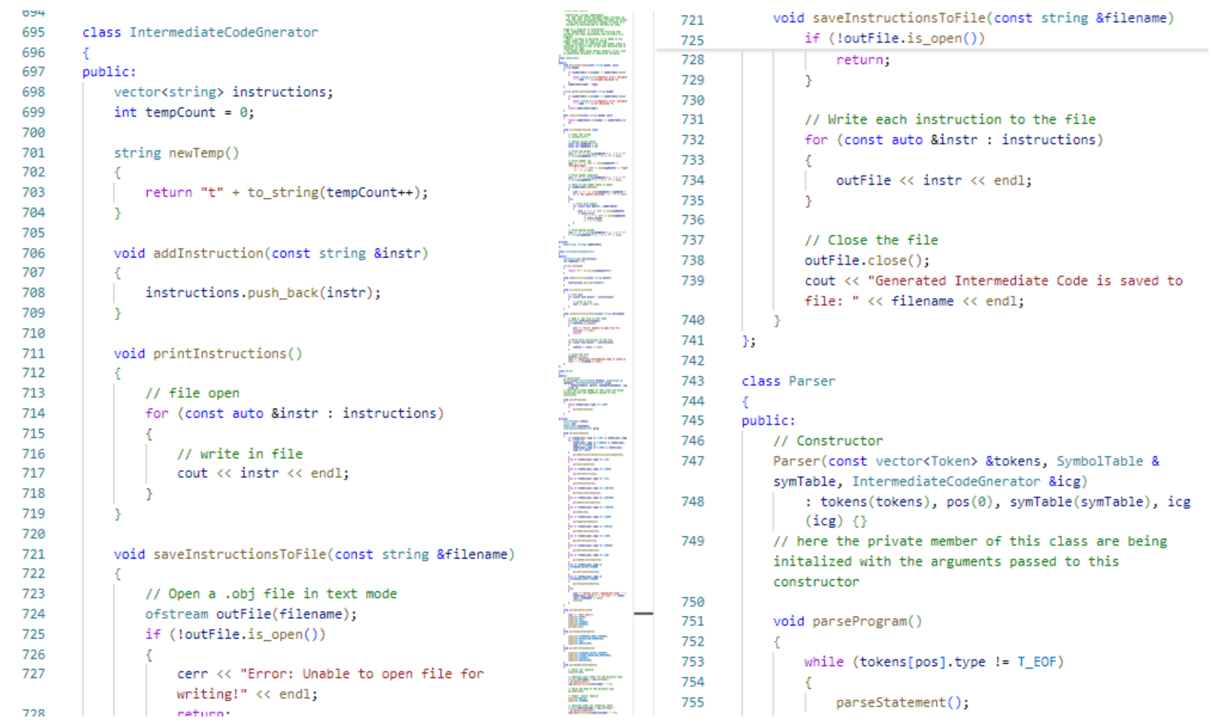
```
codeGen.saveInstructionsToFile("intermediate_code.obj");
% Saves the intermediate code to a file
```

This example saves all the generated intermediate code instructions to a file named `intermediate_code.obj`. If the file cannot be opened, an error message is displayed.

File Output: The intermediate code is saved in a text format. For example, the contents of `intermediate_code.obj` might look like:

```
t0 = x + y
t1 = t0 * z
t2 = t1 - w
```

7.5 Code



```cpp
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()
    {
        // file open
        for (const auto &instr : instructions)
        {
            // write in file
            cout << instr << endl;
        }
    }

    void saveInstructionsToFile(const string &filename)
    {
        // Open a .obj file in text mode
        ofstream outFile(filename);
        if (!outFile.is_open())
        {
            cerr << "Error: Unable to open file for
            writing!" << endl;
            return;
```

```cpp
    void saveInstructionsToFile(const string &filename)
        if (!outFile.is_open())
            return;
        }

        // Write each instruction to the file
        for (const auto &instr : instructions)
        {
            outFile << instr << endl;
        }

        // Close the file
        outFile.close();
        cout << "Generated Intermediate Code is saved to
        file: " << filename << endl;
    }
};

class Parser
{
public:
    // Constructor
    Parser(const vector<Token> &tokens, SymbolTable &
    symTable, IntermediateCodeGnerator &icg)
        : tokens(tokens), pos(0), symTable(symTable), icg
        (icg) {}
    // 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();
```

Chapter 8

Assembly Code Generation

8.1 Description

The **Code Generation** phase translates the optimized intermediate code into machine-level code or assembly code. This is the final step where the platform-specific instructions are created.

8.2 How to Implement in C++

1. Map IR instructions to machine-level instructions using a code generator.
2. Handle platform-specific details like instruction set architecture (ISA) and registers.
3. Allocate registers and memory efficiently.

8.3 Outcomes

- Platform-specific assembly or machine code.
- Errors related to unsupported operations or overflow.

8.4 Functionalities Added

8.4.1 Main Functions

- `generateAssembly(const vector<string> &tacInstructions)`: This function processes the TAC instructions and generates the corresponding assembly code. It does so by categorizing the instructions (assignment, conditional, goto, labels) and handling each type accordingly.
- `printAssembly() const`: This function prints the generated assembly code to the standard output, line by line.
- `saveInstructionsToFile(const string &filename)`: This function saves the generated assembly code to a specified file. If the file cannot be opened, an error message is displayed.

- `collectVariables(const vector<string> &instructions)`: This private function collects and declares all the variables used in the TAC instructions. It ensures that variables are declared in the `.data` section of the assembly code.

- `extractVariablesFromInstruction(const string &instr)`: This private function extracts variable names from each TAC instruction. It adds variables to the `definedVariables` set, skipping numeric constants and operators.

- `isNumeric(const string &str)`: This private function checks if a given string is a numeric value. It is used to differentiate between numeric constants and variable names.

- `processAssignment(const string &instr)`: This private function processes assignment instructions (e.g., `a = b + c`) and generates the corresponding assembly code. It handles binary operations such as addition, subtraction, multiplication, and division.

- `translateBinaryOp(const string &lhs, const string &rhs, const string &op)`: This private function translates binary operations (addition, subtraction, multiplication, division) into the corresponding assembly instructions. It handles the operand extraction and performs the operation using registers.

- `processConditional(const string &instr)`: This private function processes conditional instructions, supporting both `if` and `agar` (if in Urdu) conditions. It generates assembly code that compares values and performs conditional jumps (e.g., `if (a > b) goto label`).

- `processLabel(const string &instr)`: This private function processes label instructions (e.g., `label:`). It adds the label to the assembly code.

- `processGoto(const string &instr)`: This private function processes goto instructions (e.g., `goto label`). It generates a jump instruction in the assembly code.

- `addProgramExit()`: This private function adds a program exit section to the assembly code. It uses a system call to exit the program gracefully.

8.4.2 Key Concepts and Instruction Types

- **Assignment**: `a = b + c` is translated into assembly code where values of `b` and `c` are loaded into registers, the operation is performed, and the result is stored back into `a`.

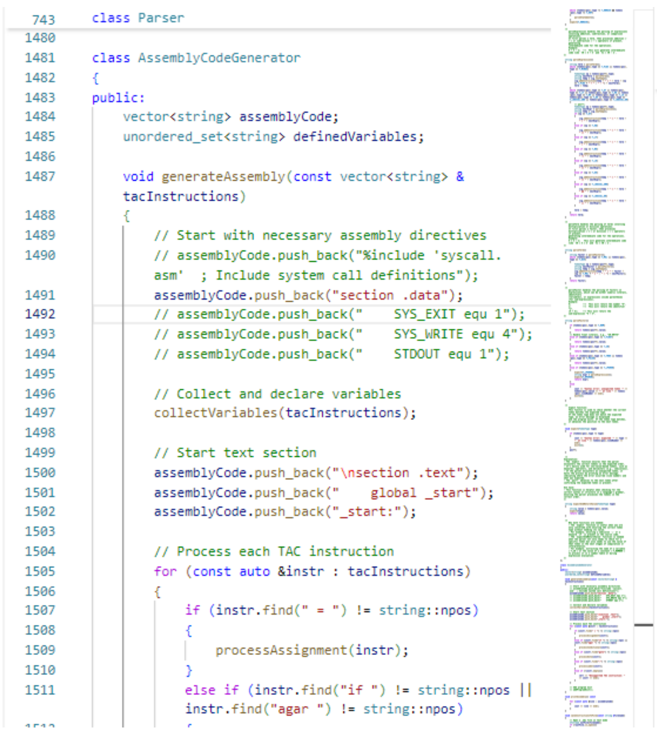
- **Binary Operations**: Operations like addition, subtraction, multiplication, and division are handled by generating the corresponding assembly instructions such as `add`, `sub`, `imul`, and `idiv`.

- **Conditionals and Goto**: The `if` and `agar` (conditional) instructions are translated into comparison and jump instructions, such as `cmp`, `je`, `jg`, `jl`, etc., to control the flow based on conditions.

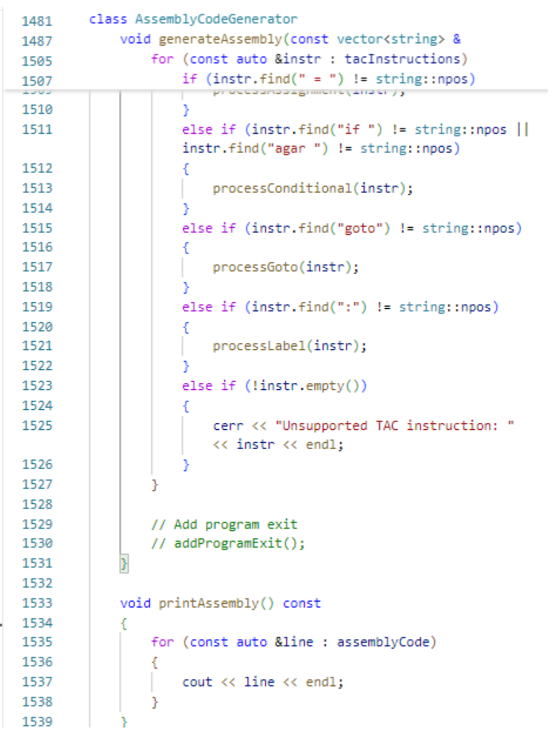
- **Labels**: Labels are inserted into the assembly code to mark specific points in the code that can be targeted by jump instructions.

In summary, the `AssemblyCodeGenerator` class is designed to translate high-level TAC instructions into low-level assembly code, ensuring that all variables are declared and used properly, while also generating the necessary control flow instructions for the program's logic.

8.5 Code



```
class Parser

class AssemblyCodeGenerator
{
public:
    vector<string> assemblyCode;
    unordered_set<string> definedVariables;

    void generateAssembly(const vector<string> &
    tacInstructions)
    {
        // Start with necessary assembly directives
        // assemblyCode.push_back("%include 'syscall.
        asm'  ; Include system call definitions");
        assemblyCode.push_back("section .data");
        // assemblyCode.push_back("    SYS_EXIT equ 1");
        // assemblyCode.push_back("    SYS_WRITE equ 4");
        // assemblyCode.push_back("    STDOUT equ 1");

        // Collect and declare variables
        collectVariables(tacInstructions);

        // Start text section
        assemblyCode.push_back("\nsection .text");
        assemblyCode.push_back("    global _start");
        assemblyCode.push_back("_start:");

        // Process each TAC instruction
        for (const auto &instr : tacInstructions)
        {
            if (instr.find(" = ") != string::npos)
            {
                processAssignment(instr);
            }
            else if (instr.find("if ") != string::npos ||
            instr.find("agar ") != string::npos)
```

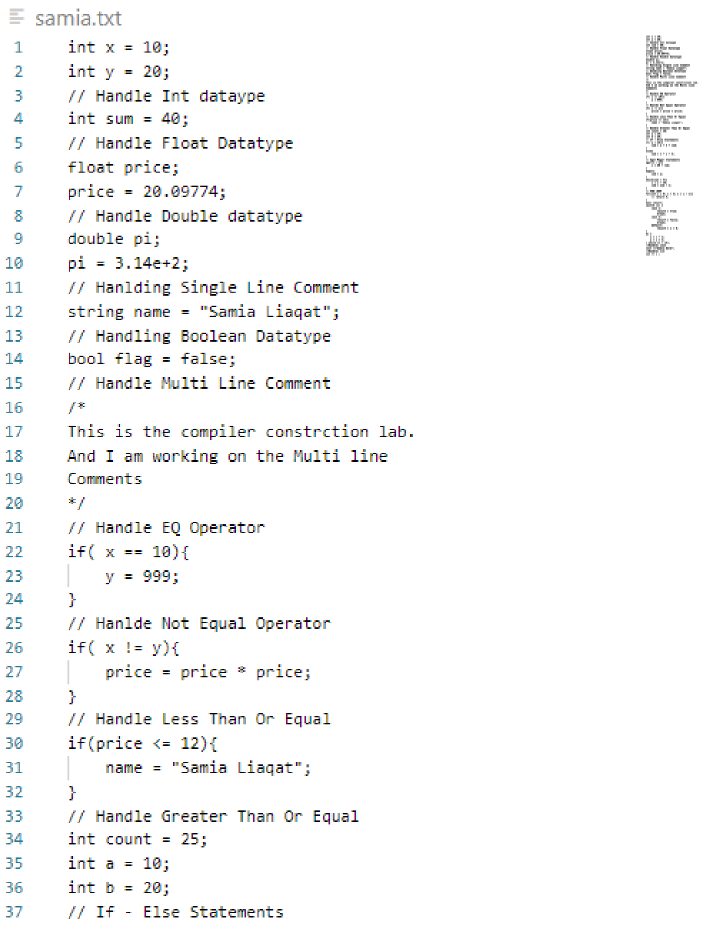


```
class AssemblyCodeGenerator
    void generateAssembly(const vector<string> &
        for (const auto &instr : tacInstructions)
            if (instr.find(" = ") != string::npos)
            }
            else if (instr.find("if ") != string::npos ||
            instr.find("agar ") != string::npos)
            {
                processConditional(instr);
            }
            else if (instr.find("goto") != string::npos)
            {
                processGoto(instr);
            }
            else if (instr.find(":") != string::npos)
            {
                processLabel(instr);
            }
            else if (!instr.empty())
            {
                cerr << "Unsupported TAC instruction: "
                << instr << endl;
            }
        }

        // Add program exit
        // addProgramExit();
    }

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

Chapter 9

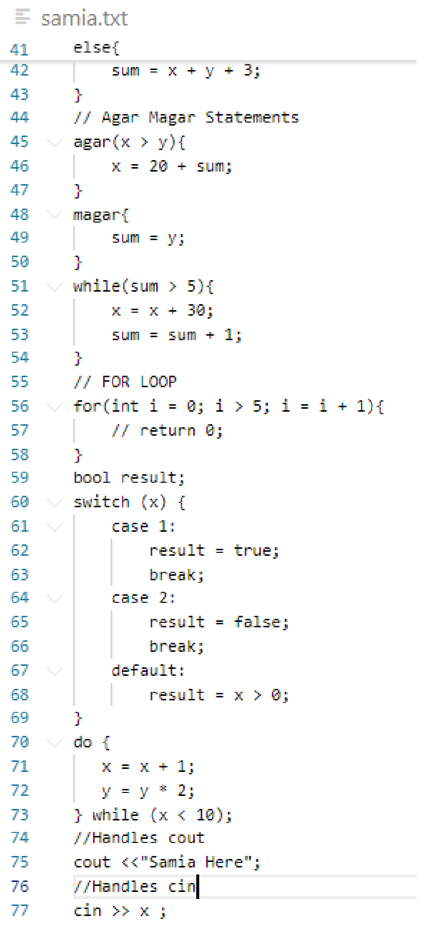
Output Screenshots of Each Phase

9.1 Test File



samia.txt

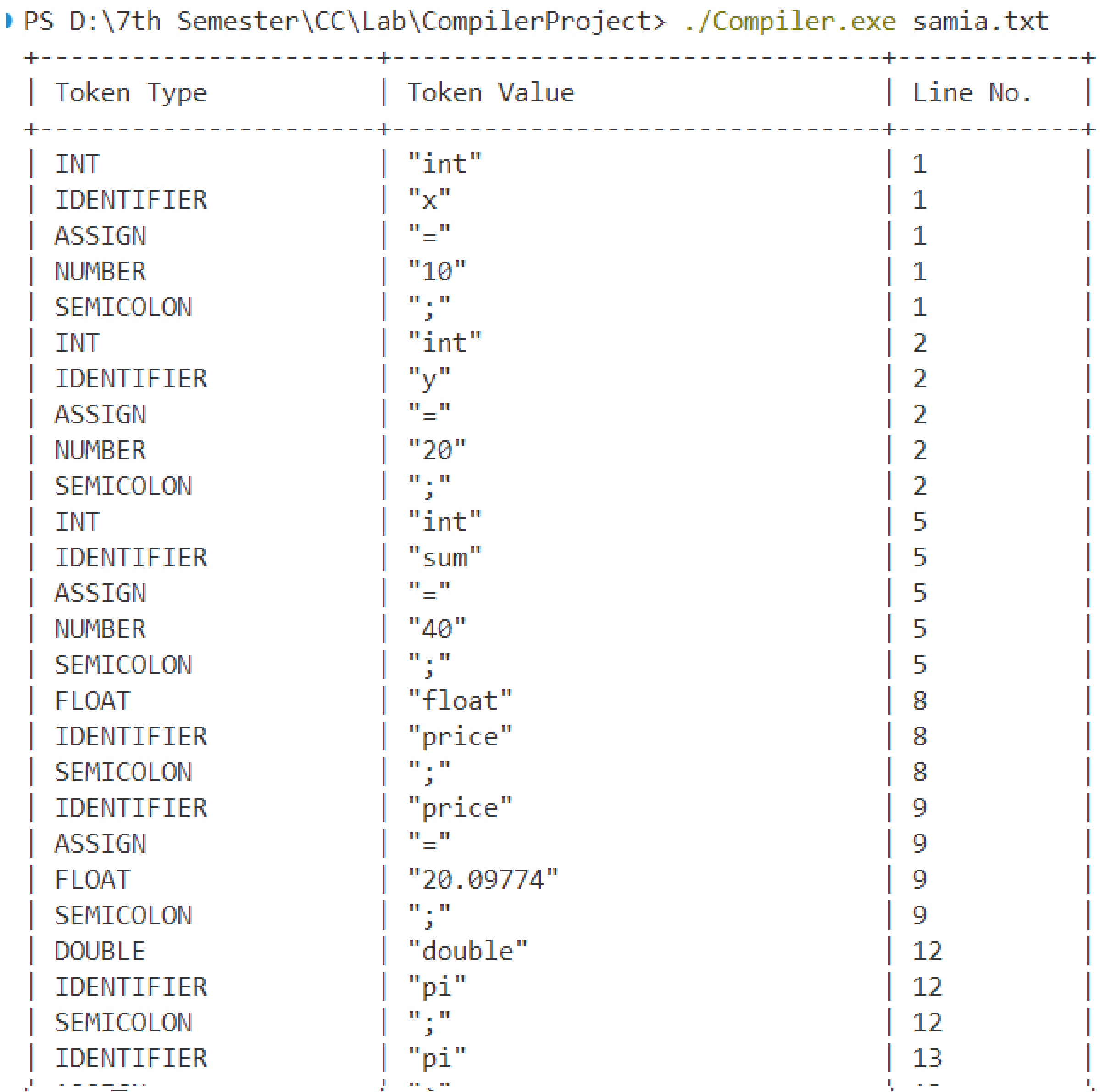
```
int x = 10;
int y = 20;
// Handle Int dataype
int sum = 40;
// Handle Float Datatype
float price;
price = 20.09774;
// Handle Double datatype
double pi;
pi = 3.14e+2;
// Hanlding Single Line Comment
string name = "Samia Liaqat";
// Handling Boolean Datatype
bool flag = false;
// Handle Multi Line Comment
/*
This is the compiler constrction lab.
And I am working on the Multi line
Comments
*/
// Handle EQ Operator
if( x == 10){
    y = 999;
}
// Hanlde Not Equal Operator
if( x != y){
    price = price * price;
}
// Handle Less Than Or Equal
if(price <= 12){
    name = "Samia Liaqat";
}
// Handle Greater Than Or Equal
int count = 25;
int a = 10;
int b = 20;
// If - Else Statements
```



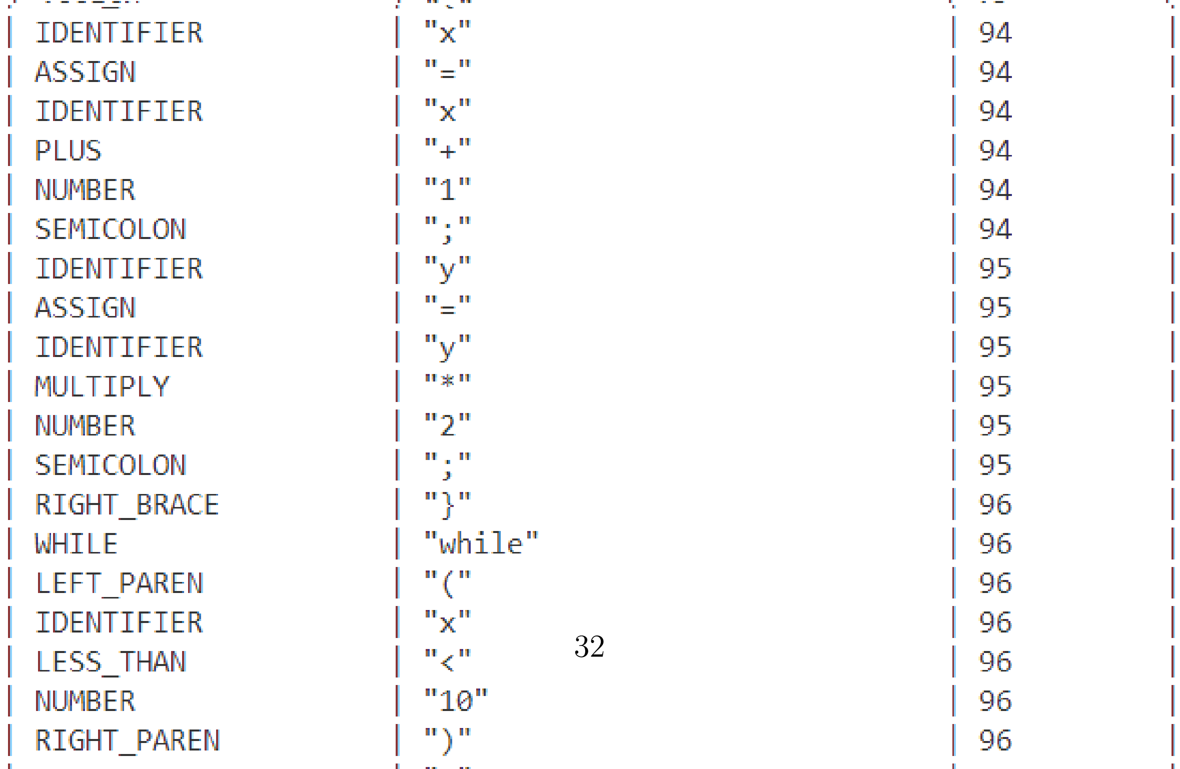
samia.txt

```
else{
    sum = x + y + 3;
}
// Agar Magar Statements
agar(x > y){
    x = 20 + sum;
}
magar{
    sum = y;
}
while(sum > 5){
    x = x + 30;
    sum = sum + 1;
}
// FOR LOOP
for(int i = 0; i > 5; i = i + 1){
    // return 0;
}
bool result;
switch (x) {
    case 1:
        result = true;
        break;
    case 2:
        result = false;
        break;
    default:
        result = x > 0;
}
do {
   x = x + 1;
   y = y * 2;
} while (x < 10);
//Handles cout
cout <<"Samia Here";
//Handles cin
cin >> x ;
```

9.2 Phase 1: Lexical Analyzer

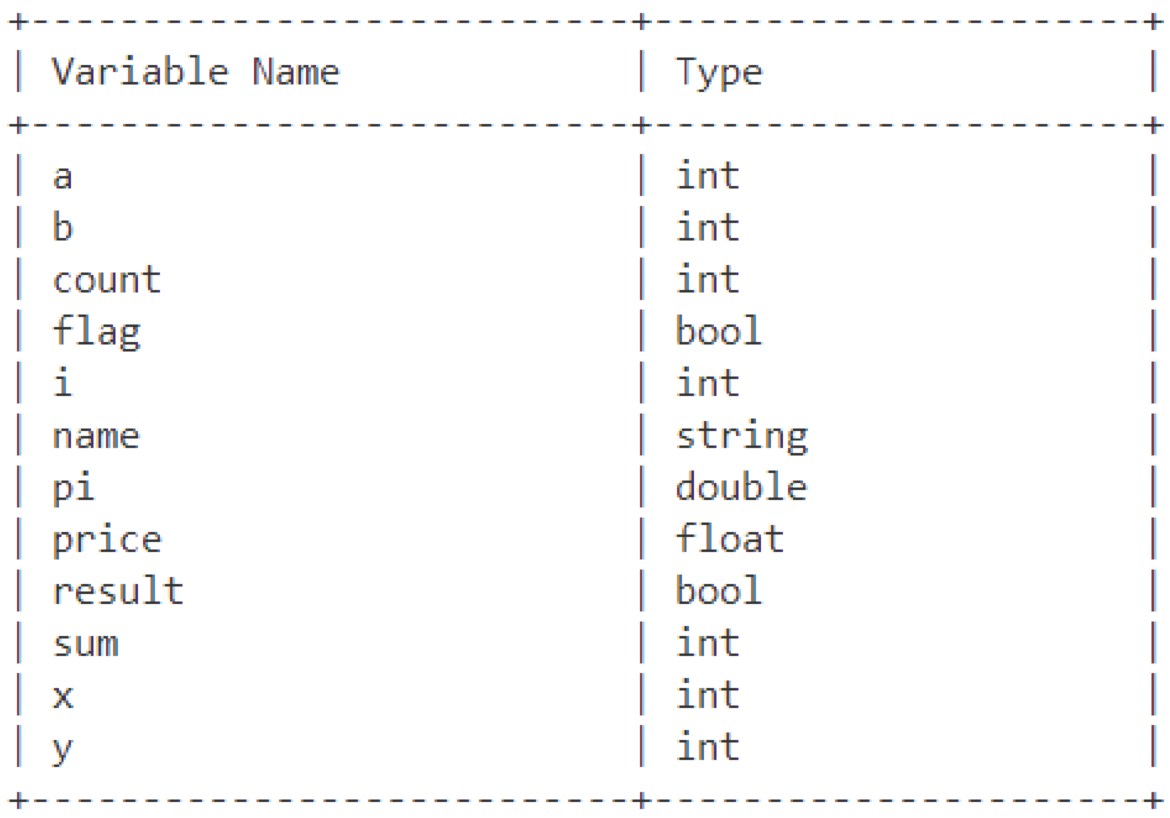


```
PS D:\7th Semester\CC\Lab\CompilerProject> ./Compiler.exe samia.txt
+-----------------------+---------------------------------+------------+
| Token Type            | Token Value                     | Line No.   |
+-----------------------+---------------------------------+------------+
| INT                   | "int"                           | 1          |
| IDENTIFIER            | "x"                             | 1          |
| ASSIGN                | "="                             | 1          |
| NUMBER                | "10"                            | 1          |
| SEMICOLON             | ";"                             | 1          |
| INT                   | "int"                           | 2          |
| IDENTIFIER            | "y"                             | 2          |
| ASSIGN                | "="                             | 2          |
| NUMBER                | "20"                            | 2          |
| SEMICOLON             | ";"                             | 2          |
| INT                   | "int"                           | 5          |
| IDENTIFIER            | "sum"                           | 5          |
| ASSIGN                | "="                             | 5          |
| NUMBER                | "40"                            | 5          |
| SEMICOLON             | ";"                             | 5          |
| FLOAT                 | "float"                         | 8          |
| IDENTIFIER            | "price"                         | 8          |
| SEMICOLON             | ";"                             | 8          |
| IDENTIFIER            | "price"                         | 9          |
| ASSIGN                | "="                             | 9          |
| FLOAT                 | "20.09774"                      | 9          |
| SEMICOLON             | ";"                             | 9          |
| DOUBLE                | "double"                        | 12         |
| IDENTIFIER            | "pi"                            | 12         |
| SEMICOLON             | ";"                             | 12         |
| IDENTIFIER            | "pi"                            | 13         |
```



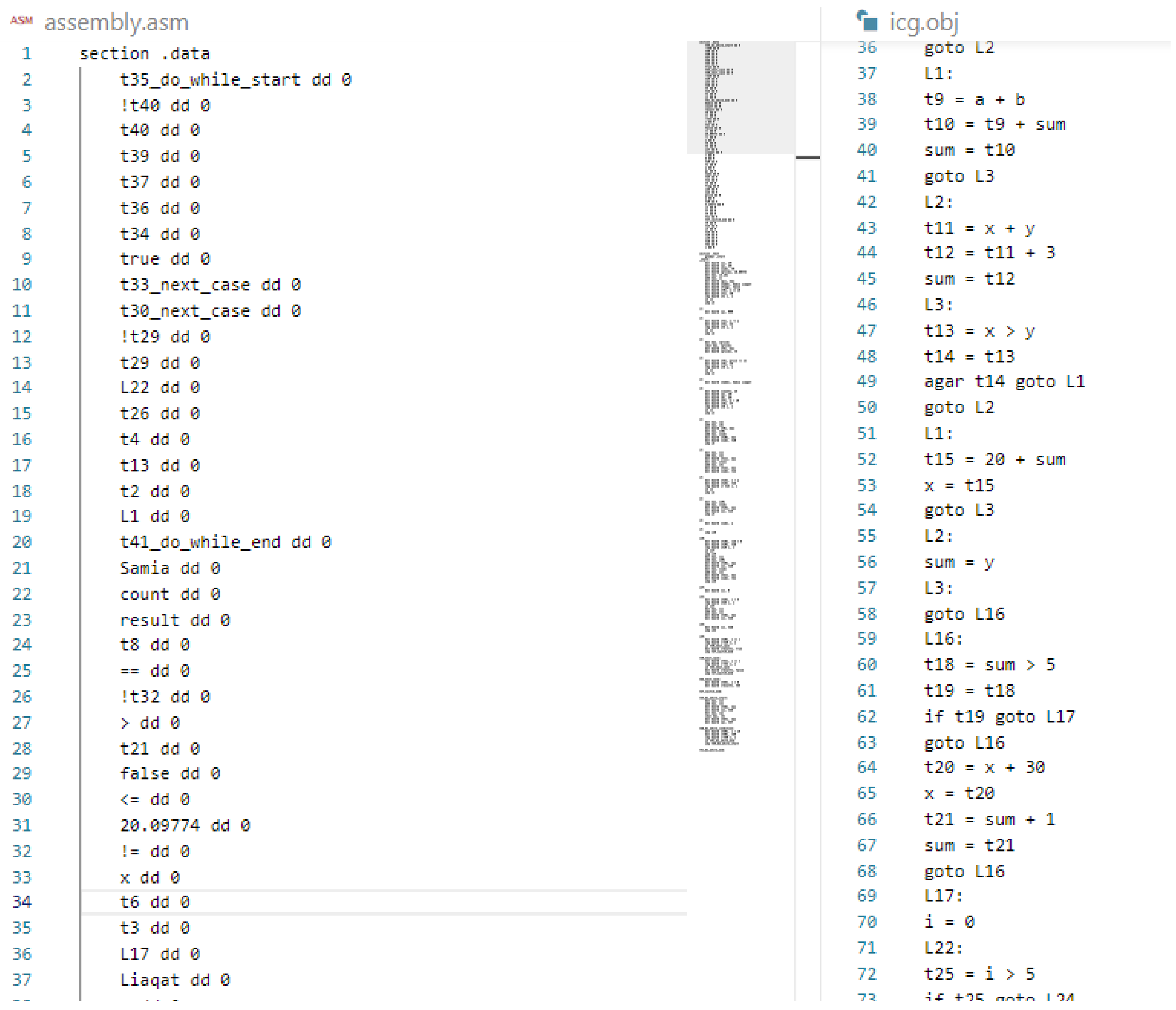
```
| IDENTIFIER            | "x"                             | 94         |
| ASSIGN                | "="                             | 94         |
| IDENTIFIER            | "x"                             | 94         |
| PLUS                  | "+"                             | 94         |
| NUMBER                | "1"                             | 94         |
| SEMICOLON             | ";"                             | 94         |
| IDENTIFIER            | "y"                             | 95         |
| ASSIGN                | "="                             | 95         |
| IDENTIFIER            | "y"                             | 95         |
| MULTIPLY              | "*"                             | 95         |
| NUMBER                | "2"                             | 95         |
| SEMICOLON             | ";"                             | 95         |
| RIGHT_BRACE           | "}"                             | 96         |
| WHILE                 | "while"                         | 96         |
| LEFT_PAREN            | "("                             | 96         |
| IDENTIFIER            | "x"                             | 96         |
| LESS_THAN             | "<"                             | 96         |
| NUMBER                | "10"                            | 96         |
| RIGHT_PAREN           | ")"                             | 96         |
```

9.3 Phase 3: Semantic Analyzer(Symbol Table)



```
+---------------------------+----------------------+
| Variable Name             | Type                 |
+---------------------------+----------------------+
| a                         | int                  |
| b                         | int                  |
| count                     | int                  |
| flag                      | bool                 |
| i                         | int                  |
| name                      | string               |
| pi                        | double               |
| price                     | float                |
| result                    | bool                 |
| sum                       | int                  |
| x                         | int                  |
| y                         | int                  |
+---------------------------+----------------------+
```

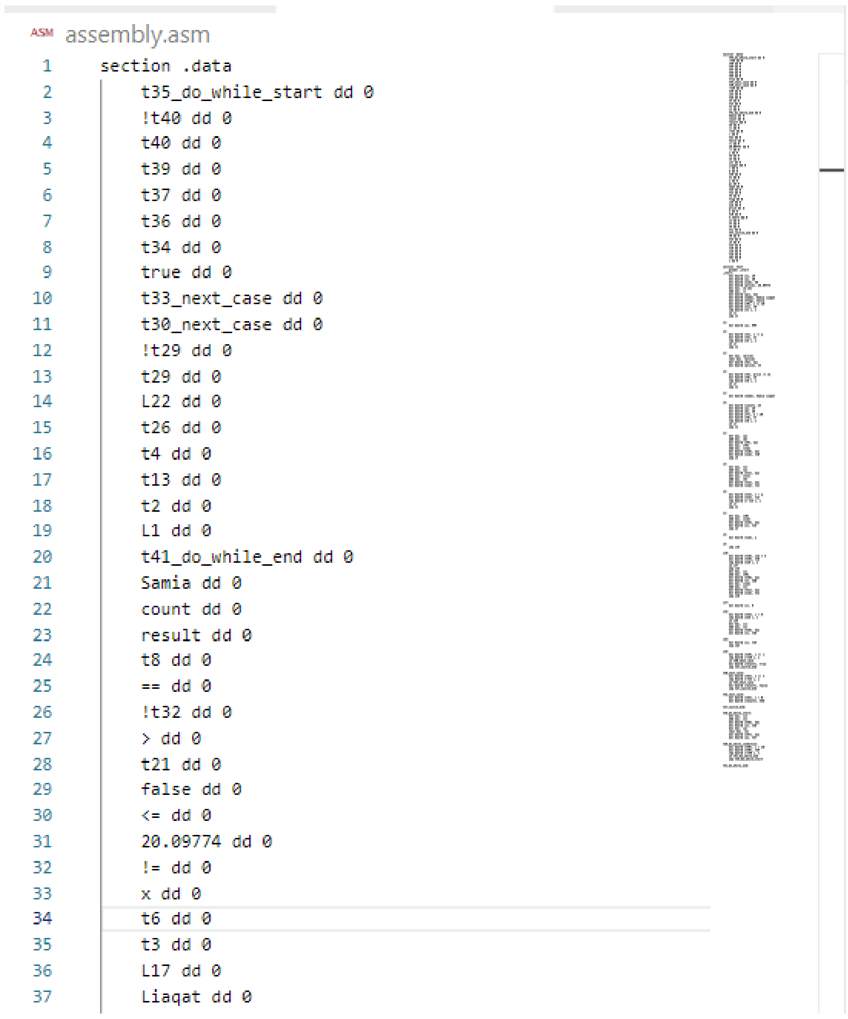
9.4 Phase 4: Intermediate Code Generation



```
section .data
    t35_do_while_start dd 0
    !t40 dd 0
    t40 dd 0
    t39 dd 0
    t37 dd 0
    t36 dd 0
    t34 dd 0
    true dd 0
    t33_next_case dd 0
    t30_next_case dd 0
    !t29 dd 0
    t29 dd 0
    L22 dd 0
    t26 dd 0
    t4 dd 0
    t13 dd 0
    t2 dd 0
    L1 dd 0
    t41_do_while_end dd 0
    Samia dd 0
    count dd 0
    result dd 0
    t8 dd 0
    == dd 0
    !t32 dd 0
    > dd 0
    t21 dd 0
    false dd 0
    <= dd 0
    20.09774 dd 0
    != dd 0
    x dd 0
    t6 dd 0
    t3 dd 0
    L17 dd 0
    Liaqat dd 0
```

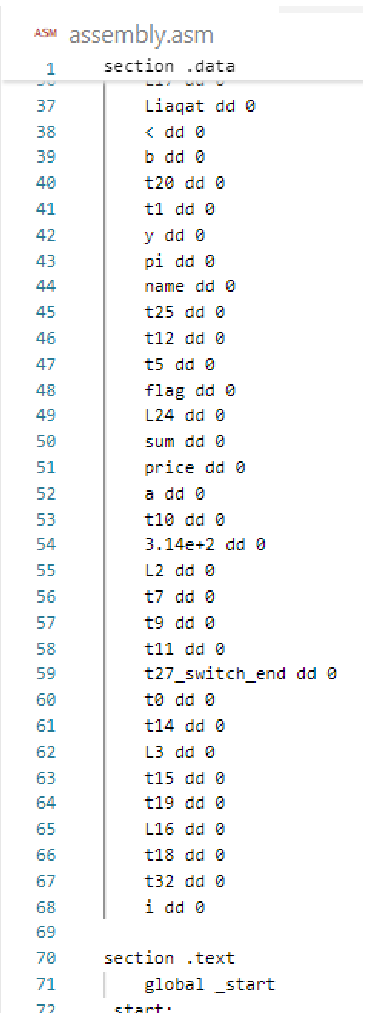
```
goto L2
L1:
t9 = a + b
t10 = t9 + sum
sum = t10
goto L3
L2:
t11 = x + y
t12 = t11 + 3
sum = t12
L3:
t13 = x > y
t14 = t13
agar t14 goto L1
goto L2
L1:
t15 = 20 + sum
x = t15
goto L3
L2:
sum = y
L3:
goto L16
L16:
t18 = sum > 5
t19 = t18
if t19 goto L17
goto L16
t20 = x + 30
x = t20
t21 = sum + 1
sum = t21
goto L16
L17:
i = 0
L22:
t25 = i > 5
if t25 goto L24
```

9.5 Phase 5: Assembly Code Generation



assembly.asm

```
section .data
    t35_do_while_start dd 0
    !t40 dd 0
    t40 dd 0
    t39 dd 0
    t37 dd 0
    t36 dd 0
    t34 dd 0
    true dd 0
    t33_next_case dd 0
    t30_next_case dd 0
    !t29 dd 0
    t29 dd 0
    L22 dd 0
    t26 dd 0
    t4 dd 0
    t13 dd 0
    t2 dd 0
    L1 dd 0
    t41_do_while_end dd 0
    Samia dd 0
    count dd 0
    result dd 0
    t8 dd 0
    == dd 0
    !t32 dd 0
    > dd 0
    t21 dd 0
    false dd 0
    <= dd 0
    20.09774 dd 0
    != dd 0
    x dd 0
    t6 dd 0
    t3 dd 0
    L17 dd 0
    Liaqat dd 0
```



assembly.asm

```
section .data
    Liaqat dd 0
    < dd 0
    b dd 0
    t20 dd 0
    t1 dd 0
    y dd 0
    pi dd 0
    name dd 0
    t25 dd 0
    t12 dd 0
    t5 dd 0
    flag dd 0
    L24 dd 0
    sum dd 0
    price dd 0
    a dd 0
    t10 dd 0
    3.14e+2 dd 0
    L2 dd 0
    t7 dd 0
    t9 dd 0
    t11 dd 0
    t27_switch_end dd 0
    t0 dd 0
    t14 dd 0
    L3 dd 0
    t15 dd 0
    t19 dd 0
    L16 dd 0
    t18 dd 0
    t32 dd 0
    i dd 0

section .text
    global _start
```

Chapter 10

Feasibility and Future Scope

The development of new programming languages is increasingly leaning towards greater similarity with natural languages to enhance user-friendliness. With the advancement of technology, languages like Python and Ruby are designed to require fewer lines of code compared to C and C++. Platforms such as Android Studio and Qt facilitate easy graphical user interface (GUI) creation while using languages like Java and C++. This project has the potential to evolve into a user-friendly, efficient, and feature-rich programming language with the attributes of an excellent coding tool.

A compiler is a software tool that automatically converts source code into machine code that can be executed by a computer. The input language is typically human-readable and not directly executable, while the output language is machine code, which the computer can understand. Although compilers are not necessary for program execution, they act as intermediaries to convert high-level programs into machine-executable code.

As technology progresses, the role of compilers is evolving to optimize code performance, scalability, and portability. They will continue to play a critical role in adapting to new programming paradigms and hardware configurations, paving the way for more efficient and powerful applications.

Chapter 11

Conclusion

In compiler design, the generation of intermediate code is a machine-independent process, ensuring that the compiler can work across various hardware platforms. This is crucial as it decouples the source code from the specific characteristics of the target machine. The subsequent transformation of intermediate code into target code is language-independent, meaning it abstracts away the complexities of the specific programming language syntax or semantics, focusing on the target architecture.

This process marks the completion of the compilation's front end, which encompasses three main stages: lexical analysis, syntax analysis, and semantic analysis. Each of these stages plays a critical role in ensuring the correctness and efficiency of the compiled code. Lexical analysis breaks down the source code into tokens, syntax analysis verifies the structure of these tokens, and semantic analysis checks for meaningfulness within the context of the programming language.

Following these analyses, intermediate code is generated, which serves as a bridge between the high-level source code and the low-level assembly code. The final stage involves converting this intermediate code into assembly code, which is platform-specific. This entire compilation process, from source code to object code, involves translating high-level language constructs into machine-readable instructions that can be executed by the computer.

This document has provided an overview of the analysis phase in compiler construction, elaborating on how the source language is methodically converted into assembly-level language during implementation. This stage ensures that all logical and syntactic constructs of the source code are preserved while making them suitable for efficient execution on target machines. By breaking down the compilation process into these distinct yet interrelated phases, we gain a deeper understanding of how complex programming languages are ultimately translated into executable programs.

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson.

[2] Appel, A. W. (2002). *Modern Compiler Implementation in C*. Cambridge University Press.

[3] Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[4] Torczon, L., & Cooper, K. D. (2011). *Engineering a Compiler* (2nd ed.). Elsevier.

[5] Grune, D., Van Reeuwijk, K., Bal, H. E., Jacobs, C., & Langendoen, K. (2012). *Modern Compiler Design*. Springer.

[6] Gough, K. J. (2000). *Compiling for the .NET Common Language Runtime*. Prentice Hall.