Compiler Construction Lab

Session: 2021 - 2025

Submitted by:
Fatiq Hussnain
2021-CS-140

Supervised by:
Mr.Laeeq khan Niazi

Department of Computer Science
University of Engineering and Technology Lahore
Pakistan

Contents

Chapter 1

Introduction

1.1 Purpose and Motivation

The primary goal of this project is to create an Assembly Code Generator that takes intermediate code, processes it, and generates corresponding assembly language instructions. This tool aims to bridge the gap between high-level programming and machine-level execution by providing a simple yet functional code generation mechanism for a hypothetical processor.

The motivation behind this project lies in the importance of code generation in a compiler. Understanding how intermediate code is converted into assembly helps us better understand the workings of compilers, optimizations, and the low-level execution of programs.

1.2 Objectives

The objectives of this project are:

- To develop an Assembly Code Generator that translates intermediate code into assembly language instructions.
- To support various operations such as arithmetic operations, relational operations, array handling, and control flow operations.
- To implement key components of a simple compiler, including a lexer, parser, intermediate code generation, and assembly generation.

1.3 Features

The core features of the Assembly Code Generator include:

- Generation of assembly instructions for variable assignments, arithmetic operations, and relational operations.
- Support for array declarations, indexing, and element assignments.
- Basic control flow support, including conditionals (if statements) and unconditional jumps (goto).
- Efficient mapping of variables to registers using a simple register allocation strategy.

1.4 Compiler Overview

A compiler is a program that translates high-level source code into machine-level code or an intermediate representation that can later be translated to machine code. The process of compiling typically involves several phases, which include lexical analysis, parsing, intermediate code generation, optimization, and code generation.

1.5 Phases of Compiler

The phases of the compiler in this project are as follows:

- **Lexical Analysis:** Converts the source code into a list of tokens (lexemes).
- **Parsing:** Converts the list of tokens into a syntax tree or an intermediate representation.
- **Intermediate Code Generation:** Transforms the high-level representation into an intermediate form that can be easily translated into assembly code.
- **Assembly Code Generation:** Converts the intermediate code into assembly language instructions that can be executed on a processor.

Chapter 2

Implementation Details

2.1 Enums and Symbol Table

The compiler uses enumerations (‘enums‘) to define various operations and token types for better code readability and maintainability. For example, an enum is used to define the types of operations in intermediate code, such as addition, subtraction, multiplication, and division.

The symbol table is a data structure that stores information about variables, functions, arrays, and other entities in the source code. It maps variables to registers or memory locations and keeps track of their types and scopes.

2.2 Three Address Code

Three Address Code (TAC) is a form of intermediate code where each instruction consists of at most three addresses: a destination, a source, and an operation. TAC is a low-level representation of the program that is easier to generate and optimize than high-level code. For example, an arithmetic operation like ‘x = a + b‘ would be represented in TAC as:

$$t1 = a + b$$

$$x = t1$$

The three addresses here are ‘a‘, ‘b‘, and ‘x‘, with the intermediate result stored in ‘t1‘.

2.3 Lexer

The Lexer is responsible for tokenizing the input source code. It splits the source code into a sequence of tokens such as keywords, identifiers, literals, and operators while ignoring whitespace and comments. The lexer also handles syntax errors like unrecognized symbols or malformed literals.
For example, given the code:
int x = 10;

The lexer will produce the following tokens:

```
INT
IDENTIFIER
ASSIGN
INT
TERMINATOR
SEMICOLON
```

The lexer ensures that only meaningful elements of the code are passed to the parser. If an invalid symbol is encountered, it will trigger an error.

2.4 Parser

The Parser is responsible for analyzing the syntax of the source code. It takes the sequence of tokens produced by the lexer and checks whether they follow the rules of the language. If the code is syntactically correct, the parser generates Three-Address Code (TAC).

During parsing, the parser may need to interact with the symbol table to verify that variables have been properly declared and are in the correct scope. For example, when parsing a declaration like:

```
int a;
```

The parser checks if ‘a‘ has been previously declared. If ‘a‘ is not in the symbol table, the parser adds it, ensuring the symbol’s type and scope are correctly assigned.

2.4.1 Example Parser Logic for a Declaration

The following function demonstrates how a declaration is parsed:

```
void parseDeclaration() {
    string type = tokens[pos].value; // Store the type
    expect(tokens[pos].type); // Expect type (int, float, char, string)

    string id = tokens[pos].value; // Store the identifier
    expect(T_ID);

    if (tokens[pos].type == T_ASSIGN) {
        expect(T_ASSIGN);
        string value = parseExpression();
        tacList.push_back(TAC("=", value, "", id)); // Add TAC for assignment
        symbolTable.addSymbol(id, type, tokens[pos].line); // Add symbol to
            table
    }
    else {
        symbolTable.addSymbol(id, type, tokens[pos].line); // Add symbol to
            table if no assignment
    }

    expect(T_SEMICOLON);
}
```

2.4.2 Control Structures

For control structures like ‘if‘ or ‘while‘, the parser first verifies the syntax of the condition and then ensures the statements inside the block are valid. For example, parsing:

```
if (x > 10) {
    y = 5;
}
```

The parser checks the expression ‘x ¿ 10‘ and parses the assignment ‘y = 5‘, making sure that both ‘x‘ and ‘y‘ are declared in the symbol table.

2.4.3 Example Parser Logic for an If Statement

The following function demonstrates how an ‘if‘ statement is parsed:

```
void parseIfStatement() {
    expect(T_IF);
    expect(T_LPAREN);
    string condition = parseExpression();
    expect(T_RPAREN);

    if (conditionIsValid(condition)) {
        string label = generateLabel();
        tacList.push_back(TAC("if", condition, "", label)); // Generate TAC for
             condition
        expect(T_LBRACE);
        parseStatement(); // Parse the statement inside the block
        expect(T_RBRACE);
    }
    else {
        throw runtime_error("Invalid␣condition␣in␣if␣statement");
    }
}
```

2.5 Assembly Generator

The `AssemblyGenerator` class is responsible for converting Three-Address Code (TAC) into assembly-like instructions. The main tasks it handles include register allocation, arithmetic operations, assignments, function calls, and output statements. It manages registers by mapping TAC variables to registers and uses a simple counter for allocating available registers.

2.5.1 Example: Processing TAC Instructions

Consider the following TAC instructions:

- $x = 5$
- $y = 10$

- $z = x + y$

The `AssemblyGenerator` class processes these instructions as follows:

1. For the assignment $x = 5$, it allocates a register for x (e.g., $R0$) and generates the assembly instruction:

   ```
   MOV R0, 5
   ```

2. For $y = 10$, it allocates a register for y (e.g., $R1$) and generates the assembly instruction:

   ```
   MOV R1, 10
   ```

3. For $z = x + y$, it checks if both x and y are in registers. Since they are, it generates the instruction:

   ```
   ADD R2, R0, R1   ; z = x + y
   ```

   It then assigns the result to $R2$ for z.

After processing these TAC instructions, the generated assembly code would look like this:

```
MOV R0, 5
MOV R1, 10
ADD R2, R0, R1   ; z = x + y
```

2.5.2 Assembly Code Generation

The `AssemblyGenerator` class works by sequentially processing each TAC instruction and generating the corresponding assembly-like instructions. The generated code typically involves:

- **MOV**: Used for simple assignments.
- **ADD**, **SUB**, **MUL**, **DIV**: For arithmetic operations.
- **JMP**, **BNE**, **BEQ**: For conditional and unconditional jumps.
- **LABEL**: For defining labels in the code.

The assembly instructions are generated by keeping track of the available registers and ensuring that each TAC variable is mapped to an appropriate register. The `AssemblyGenerator` manages register allocation by using a simple counter and mapping the TAC variables to registers.

2.5.3 Register Allocation Example

For a set of TAC instructions:

- $a = 5$
- $b = 10$
- $c = a + b$

The `AssemblyGenerator` will allocate registers $R0$, $R1$, and $R2$ for the variables a, b, and c respectively. The corresponding assembly code would be:

```
MOV R0, 5       ; a = 5
MOV R1, 10      ; b = 10
ADD R2, R0, R1  ; c = a + b
```

Here, the TAC assignment of $c = a + b$ is translated to an ADD instruction that operates on registers $R0$ and $R1$, storing the result in $R2$.

2.5.4 Handling Control Flow and Functions

The `AssemblyGenerator` also handles control flow and function calls by generating corresponding assembly instructions. For example, a conditional jump in TAC such as:

```
if (x > 10) goto label
```

Would be translated to the assembly instruction:

```
BGT R0, R1, label
```

Where $R0$ contains the value of x and $R1$ contains the value 10.

For function calls, the assembly code would typically involve generating the `CALL` and `RETURN` instructions to handle the function's execution and return values.

2.5.5 Conclusion

The `AssemblyGenerator` plays a crucial role in converting high-level Three-Address Code into assembly-like instructions. By carefully managing register allocation and generating appropriate assembly instructions for arithmetic, assignments, and control flow, it enables the translation of intermediate code into low-level executable code. This process is key to the performance and correctness of the final compiled program.

Chapter 3

Results

3.1 Results

The implementation of the compiler has successfully converted a range of source code written in a simplified programming language into Three-Address Code (TAC) and subsequently into assembly-like instructions. The following are the key results obtained from testing various source programs:

3.1.1 Correctness of the Compiler

The compiler was able to handle basic operations such as:

- Simple arithmetic expressions (addition, subtraction, multiplication, division).
- Variable declarations and assignments.
- Conditional statements and loops.
- Function calls and returns.

For each of these operations, the compiler accurately generated the corresponding TAC and assembly code. Test cases involving different combinations of operators and control structures were parsed correctly, and the resulting assembly code behaved as expected.

3.1.2 Error Handling

The compiler demonstrated robust error handling capabilities. For example:

- **Syntax Errors:** The parser detected incorrect syntax, such as missing semicolons, undeclared variables, and incorrect operator usage.
- **Semantic Errors:** The symbol table was used effectively to catch undeclared variables, ensuring that all variables used in expressions were properly declared before use.
- **Runtime Errors:** Although the compiler itself does not handle runtime errors directly, the generation of assembly code allowed for detection of memory access violations and arithmetic errors during the execution of the assembly code in the simulated environment.

3.1.3 Performance Evaluation

The compiler showed good performance for the test programs that were compiled, with the following observations:

- **Compilation Speed:** The time taken for the compiler to process typical source code was satisfactory, even for programs with a moderate size of 100–200 lines of code.
- **Memory Usage:** Memory usage remained stable, with minimal overhead during parsing and code generation phases. The symbol table and TAC lists were efficiently managed.
- **Optimization:** While the current implementation of the compiler does not include advanced optimization techniques, the generated assembly code is functional and performs adequately for the given tasks.

3.1.4 Sample Output

Consider the following example of a source program:

```
int a = 5;
int b = 10;
int c = a + b;
```

The corresponding generated Three-Address Code (TAC) and assembly code are as follows:

Three-Address Code (TAC):

```
t1 = 5
t2 = 10
t3 = t1 + t2
```

Assembly Code:

```
MOV R0, 5        ; t1 = 5
MOV R1, 10       ; t2 = 10
ADD R2, R0, R1   ; t3 = t1 + t2
```

The resulting assembly code correctly represents the operations and can be directly executed in a simple assembly simulator.

3.1.5 Additional Features

In addition to the core functionality of the compiler, several advanced features were implemented and tested:

- **Array Handling:** The compiler supports array declarations and accesses, as demonstrated in the following example:

  ```
  int arr[5];
  arr[0] = 10;
  arr[1] = 20;
  ```

The compiler correctly parses array declarations and generates appropriate TAC and assembly code to store values in array elements.

- **Function Calls:** The compiler can handle basic function calls. The corresponding assembly code for a simple function call was successfully generated, ensuring proper argument passing and return handling.

3.1.6 Limitations and Future Work

While the current implementation of the compiler is functional and demonstrates the ability to handle basic constructs of the language, there are some areas for improvement:

- **Optimization:** The generated assembly code could be further optimized to reduce unnecessary register usage and improve performance.
- **Error Reporting:** The error messages could be more descriptive, providing the user with specific details about the error, such as the line number and type of error.
- **Extended Language Features:** Support for more complex language features such as pointers, structs, and more advanced control structures (e.g., switch statements) could be added in the future.

3.1.7 Conclusion of Results

The results of the compiler demonstrate its capability to translate a high-level source program into lower-level Three-Address Code and assembly-like instructions. The key features of the compiler, such as variable handling, function support, and array parsing, functioned as intended. Although the current version of the compiler does not include advanced optimization or error recovery mechanisms, it provides a strong foundation for further development.