

# **TACIFY (Intermediate code generator)**

## **A MINI PROJECT REPORT**

*Submitted by*

**Lagan Mehta [RA2011033010073]**

**Avieral Kaushal [RA2011033010107]**

*Under the guidance of*

**Dr. J Jeyasudha**

(Assistant Professor, Department of Computational Intelligence)

*In partial satisfaction of the requirements for the degree of*

**BACHELOR OF TECHNOLOGY**

in

**COMPUTER SCIENCE & ENGINEERING**

**with specialization in Software Engineering**



**SCHOOL OF COMPUTING**

**COLLEGE OF ENGINEERING AND TECHNOLOGY**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR – 603 203**

**May-2023**



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY  
KATTANKULATHUR – 603 203

## **BONAFIDE CERTIFICATE**

Certified that this project report “**TACIFY(Intermediate Code Generator)**” is the bonafide work of “**Lagan Mehta [RA2011033010073], Avieral Kaushal [RA2011033010107]**” of III Year/VI Sem B.Tech(CSE) who carried out the mini project work under my supervision for the course **18CSC304J- Compiler Design** in SRM Institute of Science and Technology during the academic year 2023(Even Semester).

### **SIGNATURE**

Faculty In-Charge

**Dr. J Jeyasudha**

Assistant Professor

Department of Computational Intelligence

SRM Institute of Science and Technology

Kattankulathur Campus, Chennai

### **HEAD OF THE DEPARTMENT**

**Dr. R Annie Uthra**

Professor and Head ,

Department of Computational Intelligence,

SRM Institute of Science and Technology

Kattankulathur Campus, Chennai

## **ABSTRACT**

Intermediate code generation is a crucial stage in the compilation process that involves transforming source code into an intermediate representation. This intermediate representation can be easily translated into machine code or executed directly. The primary objective of intermediate code generation is to create efficient and optimized code that is machine-independent. After lexical and syntax analysis, the intermediate code generator takes the parse tree generated by the syntax analyzer and generates an intermediate representation of the program. This intermediate representation is designed to be machine-independent, making it easier to translate into machine language.

One of the main benefits of intermediate code generation is that it allows for optimizations to be performed on the program that are not possible at the source code level. Optimizations such as common subexpression elimination, loop optimization, and dead code elimination can all be performed at the intermediate code level, leading to improved program efficiency.

There are various techniques for generating intermediate code, including three-address code, quadruples, and abstract syntax trees. Three-address code, which represents an operation with three operands, is the most commonly used technique for intermediate code generation. Quadruples, on the other hand, represent an operation with four elements - an operator, two operands, and a result, and are particularly useful for generating code for control structures such as loops and conditionals. Abstract syntax trees (ASTs) are tree-like data structures that represent the syntactic structure of a program and are useful for generating code for object-oriented programming languages such as Java or C++.

In conclusion, intermediate code generation is a crucial phase in the compilation process that plays a significant role in creating optimized and efficient code. It allows for program optimization at the intermediate code level and utilizes techniques such as three-address code, quadruples, and abstract syntax trees to generate the intermediate representation.

## TABLE OF CONTENTS

Chapter No.	Title	Page No.
	ABSTRACT	3
	TABLE OF CONTENTS	4
1.	INTRODUCTION	
	1.1 Introduction	6
	1.2 Problem Statement	6
	1.3 Objectives	7
	1.4 Need for Intermediate code Generator	7
	1.5 Requirement Specifications	8
2.	NEEDS	
	2.1 Need of Compiler during Intermediate code	9
	2.2 Limitations of ICGs	9
	2.3 Types of Attributes	10
3.	SYSTEM & ARCHITECTURAL DESIGN	
	3.1 Front-End Design	11
	3.2 Front-End Architecture Design	13
	3.3 Back-End Design	13
	3.4 Back-End Architecture Design	14
	3.5 Component Design	15

4. REQUIREMENTS	
4.1 Requirements to run the script	16
5. CODING & TESTING	
5.1 Coding	17
5.2 Testing	22
6. OUTPUT & RESULT	
6.1 Output	25
6.2 Result	26
7. CONCLUSION	27
8. REFERENCES	28

# **CHAPTER 1**

## **INTRODUCTION**

### **1.1 INTRODUCTION**

Intermediate code generation is an important phase in the compilation process, where the source code is translated into an intermediate representation that can be easily optimized and transformed before generating the final executable code. The goal of this project is to develop a program that can take as input a source code file and generate the corresponding intermediate code.

The intermediate code generated should be able to represent the high-level constructs of the source code in a way that is easy to analyse and optimize. Some of the common intermediate representations include three-address code, abstract syntax trees, and bytecode.

Overall, this project will provide an opportunity to learn about the compilation process, intermediate representations, and parsing techniques. It will also help in understanding the importance of generating an optimized intermediate code for efficient code execution.

### **1.2 PROBLEM STATEMENT**

The intermediate code generator is a crucial component in the process of compiling high-level programming languages into machine code. The primary goal of this project is to develop a software tool that takes source code written in a high-level programming language as input and generates corresponding intermediate code as output.

The intermediate code generated by the tool should be in a format that can be easily translated into machine code by a later stage in the compilation process. The tool should be able to handle various types of programming constructs, such as control flow statements, arithmetic operations, function calls, and memory management.

### 1.3 OBJECTIVES

The main objective of an intermediate code generator is to bridge the gap between high-level programming languages and machine code. Its purpose is to generate an intermediate representation of the source code that can be used as input by the later stages of the compiler to produce the final machine code.

- Bridge the gap between high-level programming languages and machine code.
- Generate an intermediate representation of the source code that can be used as input by later stages of the compiler.
- Produce code that is efficient, optimized, and semantically equivalent to the original program.
- Handle various programming constructs and translate them into a format that can be easily converted into machine code.
- Generate intermediate code that is easy to debug and analyze.
- Provide a reliable and efficient way to convert high-level programming languages into machine code.
- Maintain the behavior and semantics of the original program.
- Serve as a key component of the overall compilation process.

### 1.4 Need for Intermediate Code Generator

Intermediate code generation (ICG) is an important step in the compilation process that serves several purposes. Here are some of the key reasons why ICG is necessary:

- Machine independence: ICG is necessary to make the generated code machine-independent. The intermediate code is a representation of the source code that is more abstract and independent of the target machine's architecture. This makes it easier to generate code that can run on different machines and reduces the need for machine-specific optimizations.
- Optimization: ICG provides an opportunity for optimizing the code. The intermediate code can be optimized for factors such as memory usage, execution speed, and code size. This can significantly improve the performance of the generated code and reduce the resources required to run the program.
- Error detection: ICG is useful for detecting errors in the source code. The process of generating intermediate code involves analyzing the source code and identifying potential issues, such as type mismatches, missing declarations, and syntax errors. This can help to identify and fix errors in the code before it is compiled into machine code.
- Code reuse: ICG can help to facilitate code reuse. Once the intermediate code has been generated, it can be reused in other programs that have similar requirements. This can save time and effort in the development process and promote modularity and code organization.

Overall, intermediate code generation is necessary to generate efficient, machine-independent, and optimized code. It can help to detect errors, facilitate code reuse, and improve the overall performance of the generated code.

## 1.5 REQUIREMENTS

### Hardware Requirements:

The hardware requirements for the Code generator tool will depend on the size and complexity of the programs being optimized. At a minimum, the following hardware specifications are recommended:

- A computer system with a minimum of 4GB of RAM (8GB or more recommended)
- Sufficient hard drive space to install and run the required software tools
- A reliable internet connection for downloading and installing software packages
- 

### Software Requirements:

The Code Optimizer tool will be designed to work on multiple platforms and with various programming languages. The following software requirements have been identified:

- An operating system that supports the required software tools, such as Windows, macOS, or Linux
- A text editor or integrated development environment (IDE) for writing and editing source code
- A compiler toolchain, including a parser and lexer generator, to generate the intermediate code
- A programming language to develop the intermediate code generator (such as C++, Java, or Python)
- A debugger tool for debugging the generated intermediate code
- A version control system, such as Git, for managing source code changes and collaborating with other developers



## CHAPTER 2

### 2.1 What does a compiler need to know during Intermediate code generation?

During intermediate code generation, a compiler needs to know several things in order to produce accurate and efficient code:

- The syntax and semantics of the programming language: The compiler needs to understand the rules of the programming language being used in order to correctly translate the source code into intermediate code.
- The structure and type of the data being used: The compiler needs to know the structure and type of the data being used in the source code so that it can generate intermediate code that operates correctly on that data.
- The control flow of the program: The compiler needs to analyze the control flow of the program in order to generate intermediate code that correctly represents the program's logic.
- The available hardware resources: The compiler needs to know the available hardware resources, such as the processor and memory, in order to generate intermediate code that makes efficient use of those resources.
- The optimization goals: The compiler needs to know the optimization goals, such as minimizing code size or maximizing execution speed, in order to generate intermediate code that meets those goals.

### 2.2 Limitations of ICGs.

- Additional compilation phase: The use of an intermediate code generator adds an additional phase to the compilation process, which can increase the overall compilation time.
- Dependence on the quality of the code: The quality of the intermediate code generated by the IGC is dependent on the quality of the input source code. If the source code has errors or inconsistencies, it can result in incorrect or inefficient intermediate code.
- Debugging challenges: Debugging intermediate code can be more challenging than debugging source code since the intermediate code is not in the original programming language and can have different syntax and semantics.
- Limited error reporting: The intermediate code generated by an IGC may not provide detailed error messages, making it difficult to locate and fix errors in the source code.

- Limited optimization capabilities: While an IGC can perform basic optimizations to improve the efficiency of the generated code, it may not be able to perform more complex optimizations that can be achieved by optimizing compilers.
- Compatibility issues: The intermediate code generated by an IGC may not be compatible with all target platforms, requiring additional effort to ensure compatibility.

## **2.3 Types of Attributes**

- Type attributes: These attributes specify the data type of variables and expressions in the source code. The compiler uses this information to allocate memory and generate code that operates on the correct data type.
- Address attributes: These attributes specify the memory location of variables and expressions in the source code. The compiler uses this information to generate code that accesses and modifies the correct memory location.
- Control flow attributes: These attributes specify the control flow of the program, such as loops, conditional statements, and function calls. The compiler uses this information to generate code that correctly implements the control flow of the program.
- Register allocation attributes: These attributes specify which variables and expressions should be stored in registers rather than memory. The compiler uses this information to generate code that minimizes memory access and improves performance.
- Optimization attributes: These attributes specify optimization hints to the compiler, such as loop unrolling, function inlining, and constant folding. The compiler uses this information to generate optimized code that runs faster and uses fewer resources.

## CHAPTER 3

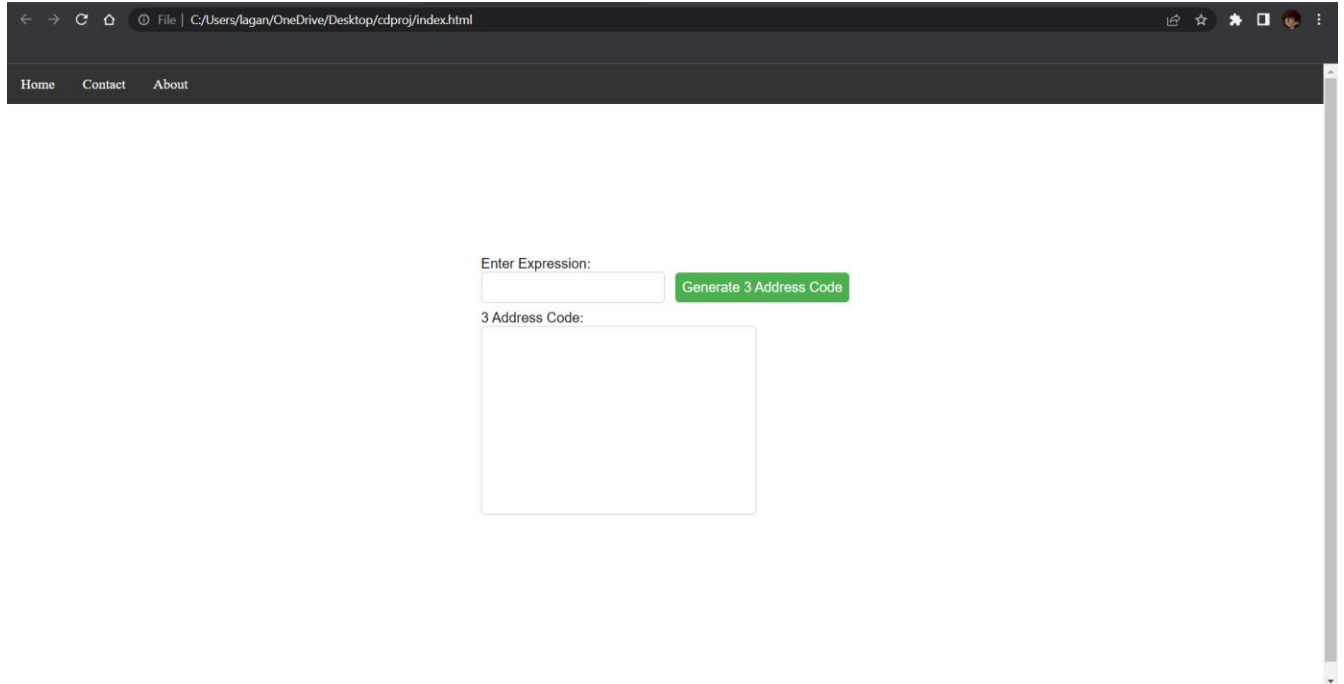
### SYSTEM ARCHITECTURE AND DESIGN

#### 3.1 FRONT-END DESIGN

For the Front-End Framework we have use Bootstrap, overview of Bootstrap

- **Bootstrap Framework:** Bootstrap is also the name of a popular front-end development framework used to build responsive and mobile-first websites. The Bootstrap framework consists of various components and tools that aid in web development, including:
- **HTML/CSS Components:** Bootstrap provides a collection of pre-designed HTML and CSS components, such as buttons, forms, navigation bars, and grids. These components can be easily integrated into web pages to ensure consistency and responsiveness.
- **JavaScript Plugins:** Bootstrap offers a set of JavaScript plugins that add functionality and interactivity to web pages. These plugins include features like carousels, modals, tooltips, and dropdown menus.
- **Responsive Grid System:** Bootstrap includes a responsive grid system that enables developers to create flexible and responsive layouts for web pages. The grid system helps in achieving a consistent look and feel across different devices and screen sizes.
- **Bootstrapping a Compiler or Interpreter:** In the context of compiler or interpreter design, bootstrapping refers to the process of implementing a compiler or interpreter for a programming language using the same language itself. The bootstrapping process typically involves the following stages:
- **Initial Compiler/Interpreter:** A basic version of the compiler or interpreter is written in a different language (often a lower-level language or an existing language). It is used to compile or interpret the subsequent versions of the compiler or interpreter.
- **Self-Compilation:** The initial compiler or interpreter is used to compile or interpret an updated version of itself written in the target language. Iterative Refinement: The process is repeated, using each new version to compile or interpret a more advanced version until the final compiler or interpreter is achieved.

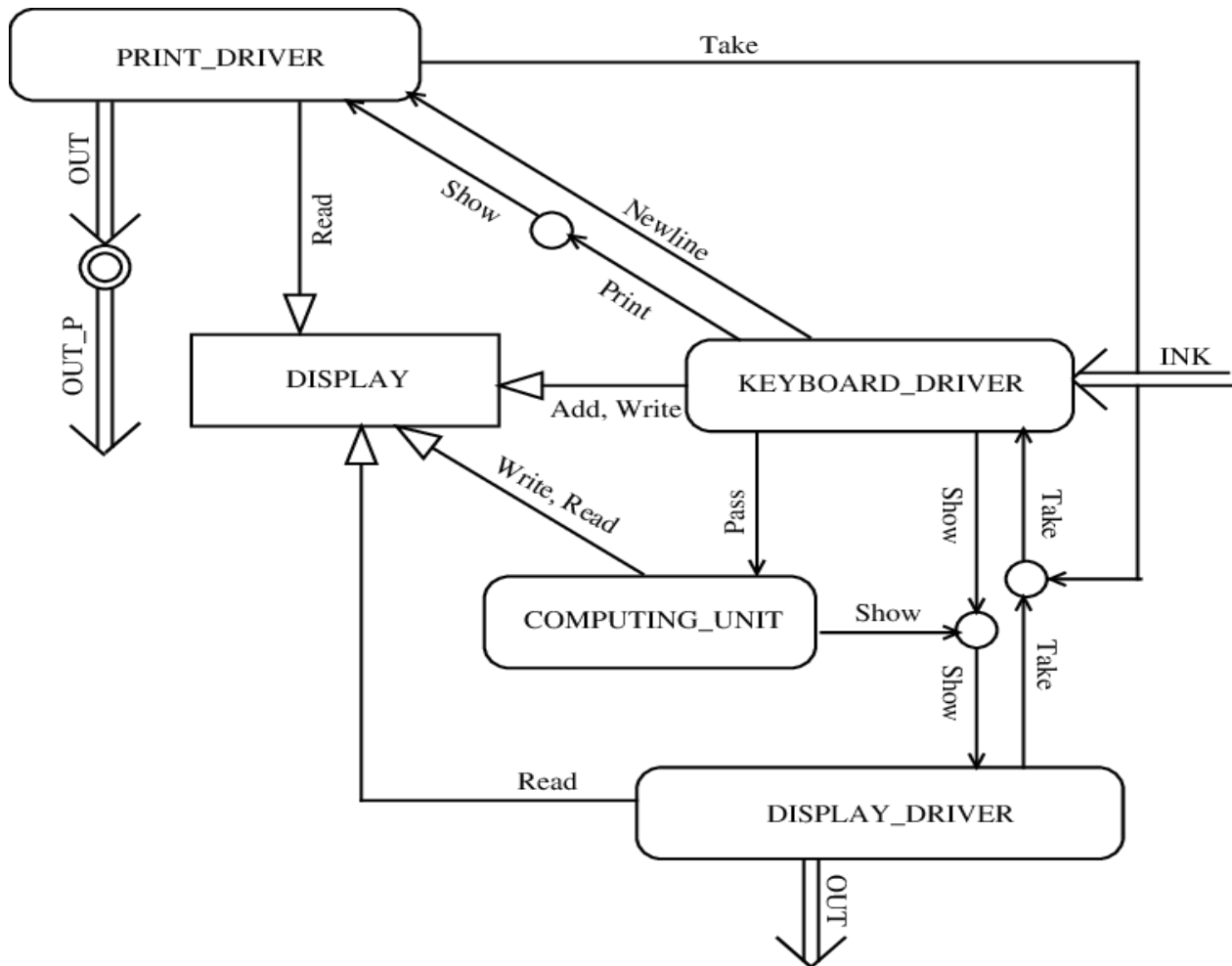
Overall, the components of bootstrap vary depending on the specific context, such as the system or software application being bootstrapped. It can involve components like the bootloader, operating system kernel, application initialization, HTML/CSS components, JavaScript plugins, and self-compilation in the case of compiler or interpreter design.



The screenshot shows a web browser window with the address bar displaying "File | C:/Users/lagan/OneDrive/Desktop/cdproj/index.html". The browser's navigation bar includes "Home", "Contact", and "About" links. The main content area features a form with the following elements:

- A label "Enter Expression:" followed by a text input field.
- A green button labeled "Generate 3 Address Code" positioned to the right of the input field.
- A label "3 Address Code:" followed by a large, empty rectangular box for the output.

### 3.2 FRONT-END ARCHITECTURE DESIGN



### 3.3 BACK-END DESIGN

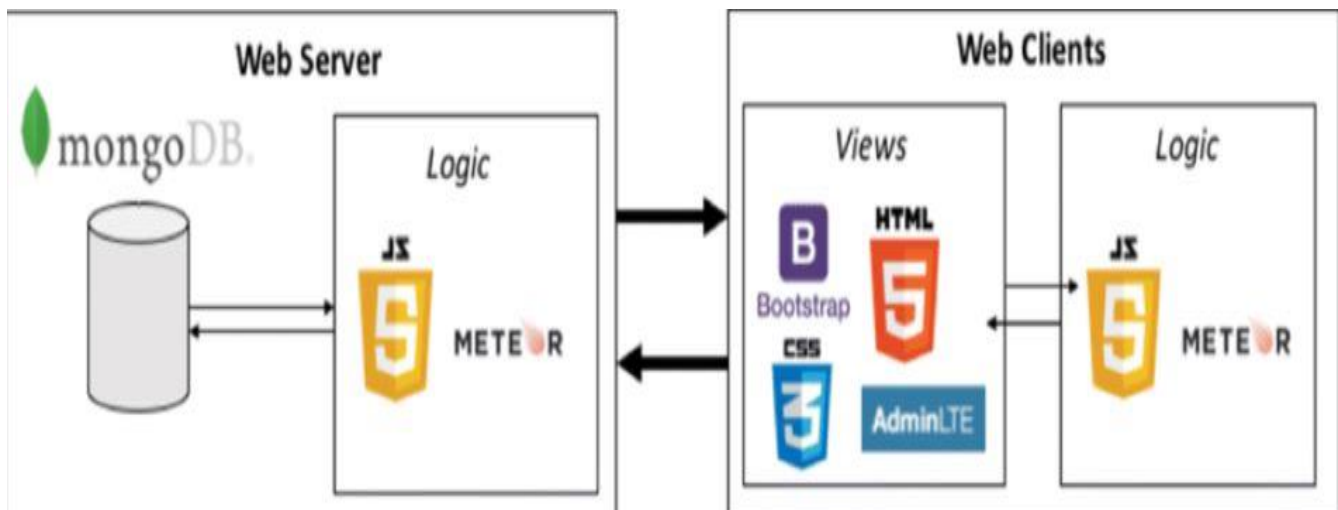
Backend design for intermediate code generation involves converting the intermediate code generated in the previous phase of the compilation process into executable code that can be run on the target machine. This process involves several stages, each of which requires careful consideration and planning. Here are some of the key aspects of backend design for intermediate code generation:

- **Instruction selection:** In this stage, the intermediate code is mapped to a set of target machine instructions. The goal is to generate code that is both correct and efficient. To accomplish this, the compiler must consider factors such as instruction set architecture, register availability, and memory access patterns.

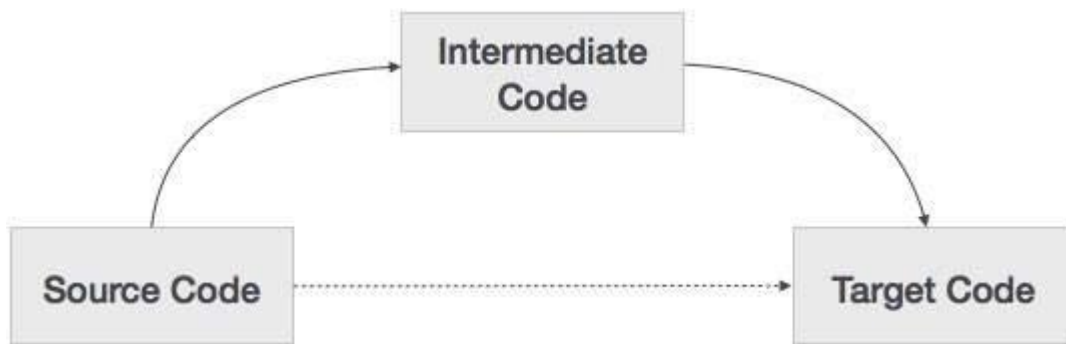
- **Register allocation:** Once the target instructions have been selected, the compiler must determine how to allocate registers for the various variables and temporary values used in the code. This is a critical optimization step, as register allocation can have a significant impact on code performance.
- **Instruction scheduling:** In this stage, the compiler determines the order in which instructions should be executed. This is done to minimize pipeline stalls, optimize cache usage, and reduce instruction dependencies.
- **Code optimization:** After the code has been scheduled and allocated to registers, the compiler applies a variety of optimization techniques to improve code performance. These may include loop unrolling, common subexpression elimination, and dead code elimination.
- **Code generation:** Finally, the optimized code is translated into machine code that can be executed on the target machine. This involves generating the appropriate machine instructions, as well as any necessary data structures, such as jump tables or function descriptors.

Overall, backend design for intermediate code generation is a complex and challenging process that requires careful consideration of many factors. By carefully optimizing each stage of the process, however, the compiler can generate code that is both correct and highly optimized for the target machine.

### 3.4 Backend/Frontend Intermediate code generator ARCHITECTURE DESIGN



### 3.5 COMPONENT DIAGRAM



## CHAPTER 4

### 4.1 The requirement to run the script

To run the script in the project, you would typically need the following requirements:

- **Compiler Infrastructure:** You would require a compiler infrastructure or framework that provides the necessary tools and libraries for parsing and code generation. Common compiler frameworks include LLVM, GCC, or custom-built compiler frameworks.
- **Programming Language:** The project's script would likely be written in a specific programming language, such as C, C++, Java, Python, or another language commonly used for compiler implementation. Ensure that you have the appropriate compiler or interpreter for the chosen programming language installed on your system.
- **Operating System:** The project should be compatible with your operating system. Most compiler development tools and frameworks are cross-platform and can be used on popular operating systems like Windows, macOS, or Linux.
- **Dependencies:** Check if the project script has any external dependencies or libraries. Make sure you have the required versions of those dependencies installed on your system. Common dependencies might include lexers/parsers (such as Lex/Yacc or ANTLR) or libraries for intermediate code representation.
- **Development Environment:** Set up a suitable development environment, such as an integrated development environment (IDE) or a text editor, to edit, compile, and run the script. Ensure that you have a proper build system (e.g., Makefile or CMake) if required.
- **Input Source Code:** The script will require input source code written in the programming language that the compiler supports. Prepare the source code files you intend to compile using the script.
- **Configuration:** Some projects may have configuration files or settings that need to be properly set up. Review any documentation or instructions provided with the project to ensure you configure it correctly.
- **System Resources:** Depending on the complexity of the script and the size of the input code, you may need sufficient system resources, such as memory and processing power, to run the compiler efficiently.



## CHAPTER 5

### CODING AND TESTING

#### 5.1 CODING

```
<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <title>3 Address Code Generator</title>
  <style>
    /* Center the page content */
    body {
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
    }

    /* Style the input and output fields */
    input, textarea {
      font-family: Arial, sans-serif;
      font-size: 16px;
      padding: 8px;
      border-radius: 5px;
      border: 1px solid #ccc;
      margin-right: 8px;
    }

    textarea {
      width: 300px;
      height: 200px;
      resize: none;
    }

    /* Style the button */
    button {
      font-family: Arial, sans-serif;
      font-size: 16px;
```

```

padding: 8px;
    border-radius: 5px;
    background-color: #4CAF50;
    color: white;
    border: none;
    cursor: pointer;
}
/* Style the output label */
label {
    font-family: Arial, sans-serif;
    font-size: 16px;
    margin-top: 8px;
    display: block;
}
ul {
list-style-type: none;
margin: 0;
padding: 0;
overflow: hidden;
background-color: #333;
position: fixed;
top: 0;
width: 100%;
}

li {
    float: left;
}

li a {
    display: block;
    color: white;
    text-align: center;
    padding: 14px 16px;
    text-decoration: none;
}
/* Change the link color to #111 (black) on hover */
li a:hover {
    background-color: #111;

```

```

    </style>
</head>
<body>

<ul>
  <li><a href="#home">Home</a></li>
  <li><a href="#contact">Contact</a></li>
  <li><a href="#about">About</a></li>
</ul>
<div>
  <label for="expression">Enter Expression:</label>
  <input type="text" id="expression">
  <button onclick="generateCode()">Generate 3 Address Code</button>
  <br>
  <label for="code">3 Address Code:</label>

<textarea id="code" readonly></textarea>
</div>
<script>
  function generate3AddressCode(expression) {
    let code = "";
    let tempCount = 0;
    let tempStack = [];
    let operatorStack = [];
    let precedence = {
      "+": 1,
      "-": 1,
      "*": 2,
      "/": 2
    };
    let assignmentTarget;

    function generateTemp() {
      let temp = "t" + tempCount;
      tempCount++;
      return temp;
    }

    function generateCodeFromStack() {
      let operator = operatorStack.pop();
      let temp2 = tempStack.pop();

```

```

let temp1 = tempStack.pop();
let temp = generateTemp();
code += temp + "=" + temp1 + " " + operator + " " + temp2 + "\n";
tempStack.push(temp);
}
for (let i = 0; i < expression.length; i++) {
let char = expression[i];
if (/[a-zA-Z]/.test(char)) {
if (!assignmentTarget) {
// This is the first variable we've seen, so it must be the assignment target
assignmentTarget = char;
} else {
tempStack.push(char);
}
} else if (/^d/.test(char)) {
let temp = generateTemp();
code += temp + "=" + char + "\n";
tempStack.push(temp);
} else if (/[\+ \- \*]/.test(char)) {
while (operatorStack.length > 0 && precedence[char] <=
precedence[operatorStack[operatorStack.length - 1]]) {
generateCodeFromStack();
}
operatorStack.push(char);
} else if (char === "(") {
operatorStack.push(char);
} else if (char === ")") {
while (operatorStack[operatorStack.length - 1] !== "(") {
generateCodeFromStack();
}
operatorStack.pop(); // Discard the "("
}
}

while (operatorStack.length > 0) {
generateCodeFromStack();
}

```

```
// If there was an assignment statement, generate the 3 address code for it
if (assignmentTarget) {
    let rhs = tempStack.pop();
    code = assignmentTarget + " = " + rhs + "\n" + code;
}

return code;
}

function generateCode() {
    let expression = document.getElementById("expression").value;
    let code = generate3AddressCode(expression);
    document.getElementById("code").value = code;
}

</script>
</body>
</html>
```

## 5.2 TESTING

### Testing the Intermediate Code Generator

A screenshot of a web browser window. The address bar shows the file path: `C:/Users/lagan/OneDrive/Desktop/cdproj/index.html`. The browser has a dark theme with navigation buttons (back, forward, refresh, home) and a menu icon. Below the address bar is a navigation bar with links: [Home](#), [Contact](#), and [About](#). The main content area is white and contains a form. The form has a label "Enter Expression:" followed by a text input field. To the right of the input field is a green button labeled "Generate 3 Address Code". Below the input field is a label "3 Address Code:" followed by a large, empty text area.

A screenshot of the same web browser window, but now the form is populated. The text input field contains the expression `x+y*z`. The green button "Generate 3 Address Code" is still visible. The large text area below now contains the following 3-address code:  
`x = t1`  
`t0 = y * z`  
`t1 = undefined + t0`

## Testing with different inputs

### INPUT 1: Priority/Operator Precedence

The screenshot shows a web browser window with the address bar displaying 'File | C:/Users/lagan/OneDrive/Desktop/cdproj/index.html#about'. The browser has a dark theme and a navigation bar with links for 'Home', 'Contact', and 'About'. The main content area contains a form with the following elements:

- A label 'Enter Expression:' above a text input field containing 'x = a+b\*c'.
- A green button labeled 'Generate 3 Address Code' to the right of the input field.
- A label '3 Address Code:' above a text area containing the following code:

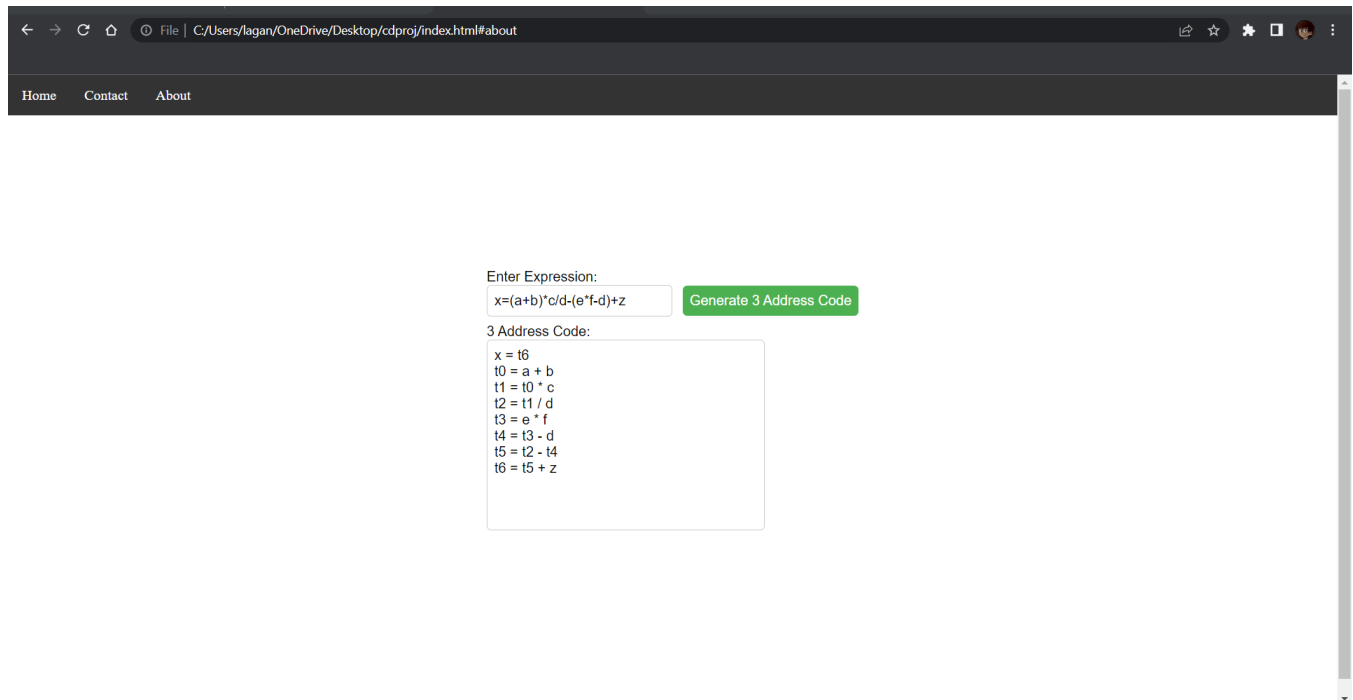
```
x = t1
t0 = b * c
t1 = a + t0
```

### INPUT 2: Braces

The screenshot shows the same web browser window as above, but with the input field containing 'x = (a+b)\*c'. The generated 3 Address Code is as follows:

```
x = t1
t0 = a + b
t1 = t0 * c
```

## INPUT 3: Large Expression



The screenshot shows a web browser window with a dark theme. The address bar displays the file path: `File | C:/Users/lagan/OneDrive/Desktop/cdproj/index.html#about`. The browser's navigation bar includes links for [Home](#), [Contact](#), and [About](#). The main content area features a form with the following elements:

- A label "Enter Expression:" above a text input field containing the expression `x=(a+b)*c/d-(e*f-d)+z`.
- A green button labeled "Generate 3 Address Code" to the right of the input field.
- A label "3 Address Code:" above a text area containing the following code:

```
x = t6
t0 = a + b
t1 = t0 * c
t2 = t1 / d
t3 = e * f
t4 = t3 - d
t5 = t2 - t4
t6 = t5 + z
```



## CHAPTER 6

### OUTPUT AND RESULTS

#### 6.1 OUTPUT:

##### Home Page

Enter Expression:

Generate 3 Address Code

3 Address Code:

```
x = t6
t0 = a + b
t1 = t0 * c
t2 = t1 / d
t3 = e * f
t4 = t3 - d
t5 = t2 - t4
t6 = t5 + z
```

##### OUTPUT

Enter Expression:

Generate 3 Address Code

3 Address Code:

```
x = t1
t0 = a + b
t1 = t0 * c
```

## **6.2 RESULT**

The result of intermediate code generation is the intermediate code that is generated by the compiler after analyzing the source code. The intermediate code is an abstract representation of the source code that is typically more machine-friendly and optimized than the original code.

The quality of the intermediate code depends on several factors, such as the design of the intermediate language, the optimization techniques used by the compiler, and the quality of the source code. A well-designed intermediate language can help to ensure that the generated code is correct and efficient. Effective optimization techniques can further improve the performance of the generated code. The intermediate code can be used as input to the next phase of the compilation process, which is typically the backend code generation phase. During this phase, the intermediate code is translated into machine code that can be executed on the target machine.

Overall, the result of intermediate code generation is an important step in the compilation process that plays a critical role in ensuring that the generated code is correct, efficient, and optimized for the target machine.

## CHAPTER 7

### 7.1 CONCLUSION

Intermediate code generation is a crucial step in the compilation process that involves translating the high-level source code into an intermediate representation that is more machine-friendly and efficient. The resulting intermediate code is an abstract representation of the source code that is typically optimized for the target machine.

During intermediate code generation, the compiler analyzes the source code and generates an intermediate representation that is typically more abstract and machine-independent than the original code. Attributes are attached to programming language constructs, such as variables and expressions, to provide additional information that is used by the compiler during intermediate code generation.

The quality of the intermediate code depends on several factors, including the design of the intermediate language, the optimization techniques used by the compiler, and the quality of the source code. Effective optimization techniques can significantly improve the performance of the generated code, while a well-designed intermediate language can ensure that the generated code is correct and efficient.

The intermediate code is typically used as input to the next phase of the compilation process, which is the backend code generation phase. During this phase, the intermediate code is translated into machine code that can be executed on the target machine.

In conclusion, intermediate code generation is a critical step in the compilation process that plays a critical role in ensuring that the generated code is correct, efficient, and optimized for the target machine. Effective intermediate code generation can significantly improve the overall performance of the generated code and contribute to the success of the software development process.

## REFERENCES

- "Intermediate Code Generation" by GeeksforGeeks -  
<https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/>
- "Intermediate Code Generation" by Tutorials Point -  
[https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_intermediate\\_code\\_generation.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generation.htm)
- "Intermediate Code Generation" by Stanford University -  
<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/07/Slides07.pdf>
- "Intermediate Code Generation" by Indian Institute of Technology Kharagpur -  
<http://cse.iitkgp.ac.in/~abhij/course/compilers/lectures/lecture11.pdf>
- "Compiler Design: Intermediate Code Generation" by YouTube channel Neso Academy -  
<https://www.youtube.com/watch?v=ZRI1p7YTkzc>