



BONAFIDE CERTIFICATE

Certified that this project report “**MINI COMPILER(Code Optimization)**” is the bonafide work of “**SANSKAR SHARMA [RA20110033010079]**” “**RISHU RAJ [RA20110033010082]**” 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).

A handwritten signature in red ink, appearing to be 'J Jeyasudha', with the year '2023' written below it.

SIGNATURE

Faculty In-Charge

Dr. J Jeyasudha

Assistant Professor

Department of Computational Intelligence

SRM Institute of Science and Technology

Kattankulathur Campus, Chennai

A handwritten signature in blue ink, appearing to be 'Dr. R Annie Uthra'.

HEAD OF THE DEPARTMENT

Dr. R Annie Uthra

Professor and Head ,

Department of Computational Intelligence,

SRM Institute of Science and Technology

Kattankulathur Campus, Chennai

MINI COMPILER(Code Optimization)

A MINI PROJECT REPORT

Submitted by

SANSKAR SHARMA[RA2011033010079]

RISHU RAJ [RA2011033010082]

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 – 603203

May-2023



BONAFIDE CERTIFICATE

Certified that this project report “**MINI COMPILER(Code Optimization)**” is the bonafide work of “**SANSKAR SHARMA [RA20110033010079]**” “**RISHU RAJ [RA20110033010082]**” 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

The Code Optimizer project aims to develop an automated tool for optimizing computer programs. The tool will analyse source code written in a variety of programming languages and apply various optimization techniques to improve its performance, reduce its memory usage, and enhance its maintainability.

The optimizations will be based on both static and dynamic analysis of the code, including profiling, data flow analysis, and machine learning. The resulting optimized code will be tested using a suite of benchmark programs to measure its effectiveness.

The ultimate goal of this project is to help developers write faster, more efficient, and more reliable software with less effort. The goal of the Code Optimizer project is to create a software tool that automatically analyzes and optimizes computer programs to make them faster, more efficient, and easier to maintain. The tool will use a combination of static and dynamic analysis techniques to identify areas of the code that can be improved and apply optimizations to these areas.

The project is being developed with the aim of helping software developers improve the performance and reliability of their programs, while also reducing the amount of time and effort required to achieve these goals.

TABLE OF CONTENTS

Chapter No.	Title	Page No.
	ABSTRACT	3
	TABLE OF CONTENTS	4
1.	INTRODUCTION	
1.1	Introduction	6
1.2	Problem Statement	7
1.3	Objectives	8
1.4	Need For Semantic Analyser	10
1.5	Requirement Specifications	12
2.	NEEDS	
2.1	Need of Compiler during Semantic Analysis	13
2.2	Limitations of CFGs	13
2.3	Types of Attributes	14
3.	SYSTEM & ARCHITECTURAL DESIGN	
3.1	Front-End Design	21
3.2	Front-End Architecture Design	23
3.3	Back-End Design	23
3.4	Back-End Architecture Design	25
3.5	Code Optimization Architecture Design	26

4. REQUIREMENTS	
4.1 Requirements to run the script	27
5. CODING & TESTING	
5.1 Coding	28
5.2 Testing	36
6. OUTPUT & RESULT	
6.1 Output	38
6.2 Result	42
7. CONCLUSION	43
8. REFERENCES	44

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Software development is a complex process that requires developers to balance multiple priorities, such as functionality, usability, and performance. One of the key challenges of software development is optimizing the performance of the code, which is essential for creating fast and efficient software. While there are many optimization techniques that developers can use to improve the performance of their code, the process can be time-consuming and require a high level of expertise.

To address these challenges, the Code Optimizer project aims to develop an automated tool that can optimize computer programs automatically. The tool will use a combination of static and dynamic analysis techniques to identify areas of the code that can be improved and apply optimizations to these areas. By automating the optimization process, the tool will reduce the amount of time and effort required to optimize code, while also improving the efficiency and reliability of software.

The Code Optimizer project has the potential to have a significant impact on the software development industry. By automating the optimization process, the tool will help developers create faster, more efficient, and more reliable software with less effort. This will enable developers to focus on other important aspects of software development, such as functionality and usability. The tool will also help to democratize optimization techniques, making them more accessible to developers with varying levels of expertise.

1.2 PROBLEM STATEMENT

The problem is that the optimization process is often done manually, which can be a tedious and error-prone task. Developers need to identify areas of the code that can be optimized and apply optimizations, such as loop unrolling, dead code elimination, and data flow analysis. This process can take a lot of time and requires a deep understanding of the code and the programming language.

Furthermore, optimization is often an iterative process that requires developers to test the code after each optimization to ensure that it still works as expected. This testing process can also be time-consuming and require a significant amount of effort. To address these challenges, the Code Optimizer project aims to develop an automated tool that can optimize computer programs automatically. The tool will use a combination of static and dynamic analysis techniques to identify areas of the code that can be improved and apply optimizations to these areas. By automating the optimization process, the tool will reduce the amount of time and effort required to optimize code, while also improving the efficiency and reliability of software. The problem statement, therefore, is to create an automated tool that can optimize computer programs automatically, reducing the time and effort required to optimize code, while also improving the efficiency and reliability of software. The tool should be able to work with a variety of programming languages and apply a range of optimization techniques to improve the performance of the code. The tool should also be tested to ensure that it can optimize code under different conditions and environments.

1.3 OBJECTIVES

The primary objective of the Code Optimizer project is to develop an automated tool that can optimize computer programs automatically, reducing the time and effort required to optimize code, while also improving the efficiency and reliability of software. To achieve this objective, the following specific objectives have been identified:

- Develop a prototype of the Code Optimizer tool that can analyze source code and identify areas that can be optimized using a combination of static and dynamic analysis techniques.
- Implement a range of optimization techniques in the Code Optimizer tool, including loop unrolling, dead code elimination, data flow analysis, and machine learning.
- Test the Code Optimizer tool using a suite of benchmark programs designed to test the performance and efficiency of the optimized code under different conditions and environments.
- Evaluate the effectiveness of the Code Optimizer tool in terms of reducing the time and effort required to optimize code, while also improving the efficiency and reliability of software.
- Develop documentation for the Code Optimizer tool, including user guides and technical documentation, to facilitate its integration into the software development process.
- Explore the potential for integrating the Code Optimizer tool into existing Integrated Development Environments (IDEs) to further streamline the optimization process for developers.
- Share the results of the Code Optimizer project with the software development community to facilitate knowledge sharing and promote the adoption of optimization techniques in software development.

1.4 Requirements : Hardware and Software

Hardware Requirements:

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

- Processor: Intel Core i5 or higher
- Memory 8 GB RAM or higher
- Storage At least 50 GB of free disk space
- Display: 1366 x 768 resolution or higher
- Internet connection: Required for downloading and updating the tool

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:

- Operating System: Windows 10, Linux, or MacOS
- Programming Languages: The tool should be able to optimize code written in various programming languages, such as C, C++, Java, Python, and others.
- Integrated Development Environment (IDE): The tool should be compatible with various IDEs, such as Visual Studio, Eclipse, NetBeans, and others.
- Compiler: The tool should be able to work with different compilers, such as GCC, Clang, and MSVC, to generate optimized code.
- Libraries: The tool should support different libraries commonly used in software development, such as standard template libraries (STL), Boost, and others.

CHAPTER 2

2.1 What does a compiler need to know during semantic analysis?

Whether a variable has been declared? Are there variables which have not been declared? What is the type of the variable? Whether a variable is a scalar, an array, or a function? What declaration of the variable does each reference use? If an expression is type consistent? If an array use like $A[i,j,k]$ is consistent with the declaration? Does it have three dimensions?

For example, we have the third question from the above list, i.e., what is the type of a variable and we have a statement like `int a, b, c;`

Then we see that syntax analyzer cannot alone handle this situation. We actually need to traverse the parse trees to find out the type of identifier and this is all done in semantic analysis phase. Purpose of listing out the questions is that unless we have answers to these questions we will not be able to write a semantic analyzer. This becomes a feedback mechanism. If the compiler has the answers to all these questions only then will it be able to successfully do a semantic analysis by using the generated parse tree. These questions give a feedback to what is to be done in the semantic analysis. These questions help in outlining the work of the semantic analyzer. In order to answer the previous questions the compiler will have to keep information about the type of variables, number of parameters in a particular function etc. It will have to do some sort of computation in order to gain this information. Most compilers keep a structure called symbol table to store this information. At times the information required is not available locally, but in a different scope altogether. In syntax analysis we used context free grammar. Here we put lot of attributes around it. So it consists of context sensitive grammars along with extended attribute grammars. Ad-hoc methods also good as there is no structure in it and the formal method is simply just too tough. So we would like to use something in between. Formalism may be so difficult that writing specifications itself may become tougher than writing compiler itself. So we do use attributes but we do analysis along with parse tree itself instead of using context sensitive grammars.

2.2 Limitations of CFGs.

Context free grammars deal with syntactic categories rather than specific words. The declare before use rule requires knowledge which cannot be encoded in a CFG and thus CFGs cannot match an instance of a variable name with another. Hence we introduce the attribute grammar framework.

Syntax directed definition

This is a context free grammar with rules and attributes. It specifies values of attributes by associating semantic rules with grammar productions.

An example

PRODUCTION	SEMANTIC RULE
$E \rightarrow E1 + T$	$E.code = E1.code$

Syntax Directed Translation

This is a compiler implementation method whereby the source language translation is completely driven by the parser.

The parsing process and parse tree are used to direct semantic analysis and translation of the source program.

Here we augment conventional grammar with information to control semantic analysis and translation.

This grammar is referred to as attribute grammar.

The two main methods for SDT are Attribute grammars and syntax directed translation scheme

Attributes

An attribute is a property whose value gets assigned to a grammar symbol. Attribute computation functions, also known as semantic functions are functions associated with productions of a grammar and are used to compute the values of an attribute. Predicate functions are functions that state some syntax and the static semantic rules of a particular grammar.

Types of Attributes

Type -These associate data objects with the allowed set of values.

Location - May be changed by the memory management routine of the operating system.

Value -These are the result of an assignment operation.

Name-These can be changed when a sub-program is called and returns.

Component - Data objects comprised of other data objects. This binding is represented by a pointer and is subsequently changed.

Synthesized Attributes.

These attributes get values from the attribute values of their child nodes. They are defined by a semantic rule associated with the production at a node such that the production has the non-terminal as its head.

An example

$S \rightarrow ABC$

S is said to be a synthesized attribute if it takes values from its child node (A, B, C).

An example

$E \rightarrow E + T \{ E.value = E.value + T.value \}$

Parent node E gets its value from its child node.

Inherited Attributes.

These attributes take values from their parent and/or siblings.

They are defined by a semantic rule associated with the production at the parent such that the production has the non-terminal in its body.

They are useful when the structure of the parse tree does not match the abstract syntax tree of the source program.

They cannot be evaluated by a pre-order traversal of the parse tree since they depend on both left and right siblings.

An example;

$S \rightarrow ABC$

A can get its values from S, B and C.

B can get its values from S, A and C

C can get its values from A, B and S

Expansion

This is when a non-terminal is expanded to terminals as per the provided grammar.

Reduction

This is when a terminal is reduced to its corresponding non-terminal as per the grammar rules.

Note that syntax trees are parsed top, down and left to right

Attribute grammar

This is a special case of context free grammar where additional information is appended to one or more non-terminals in-order to provide context-sensitive information.

We can also define it as SDDs without side-effects.

It is the medium to provide semantics to a context free grammar and it helps with the specification of syntax and semantics of a programming language.

When viewed as a parse tree, it can pass information among nodes of a tree.

An Example

Given the CFG below;

$$E \rightarrow E + T \{ E.value = E.value + T.value \}$$

The right side contains semantic rules that specify how the grammar should be interpreted.

The non-terminal values of E and T are added and their result copied to the non-terminal E.

An Example

Consider the grammar for signed binary numbers

$$\text{number} \rightarrow \text{signlist}$$
$$\text{sign} \rightarrow + \mid -$$
$$\text{list} \rightarrow \text{listbit} \mid \text{bit}$$
$$\text{bit} \rightarrow 0 \mid 1$$

We want to build an attribute grammar that annotates Number with the value it represents.

First we associate attributes with grammar symbols

SYMBOL	ATTRIBUTES
number	val
sign	neg
list	pos, val
bit	pos, val

The attribute grammar

iq.opengenus.org

Production	Attribute Rule
$number \rightarrow sign\ list$	$list.pos = 0$ if $sign.neg$: $number.val = -list.val$ else: $number.val = list.val$
$sign \rightarrow +$	$sign.neg = false$
$sign \rightarrow -$	$sign.neg = true$
$list \rightarrow bit$	$bit.pos = list.pos$ $list.val = bit.val$
$list_0 \rightarrow list_1\ bit$	$list_1.pos = list_0.pos + 1$ $bit.pos = list_0.pos$ $list_0.val = list_1.val + bit.val$
$bit \rightarrow 0$	$bit.val = 0$
$bit \rightarrow 1$	$bit.val = 2^{bit.pos}$

Defining an Attribute Grammar

Attribute grammar will consist of the following features;

- Each symbol X will have a set of attributes $A(X)$
- $A(X)$ can be;
 - Extrinsic attributes obtained outside the grammar, notable the symbol table
 - Synthesized attributes passed up the parse tree
 - Inherited attributes passed down the parse tree.
- Each production of the grammar will have a set of semantic functions and predicate functions(may be an empty set)

- Based on the way an attribute gets its value, attributes can be divided into two categories; these are, Synthesized or inherited attributes.

Abstract Syntax Trees(ASTs)

These are a reduced form of a parse tree.

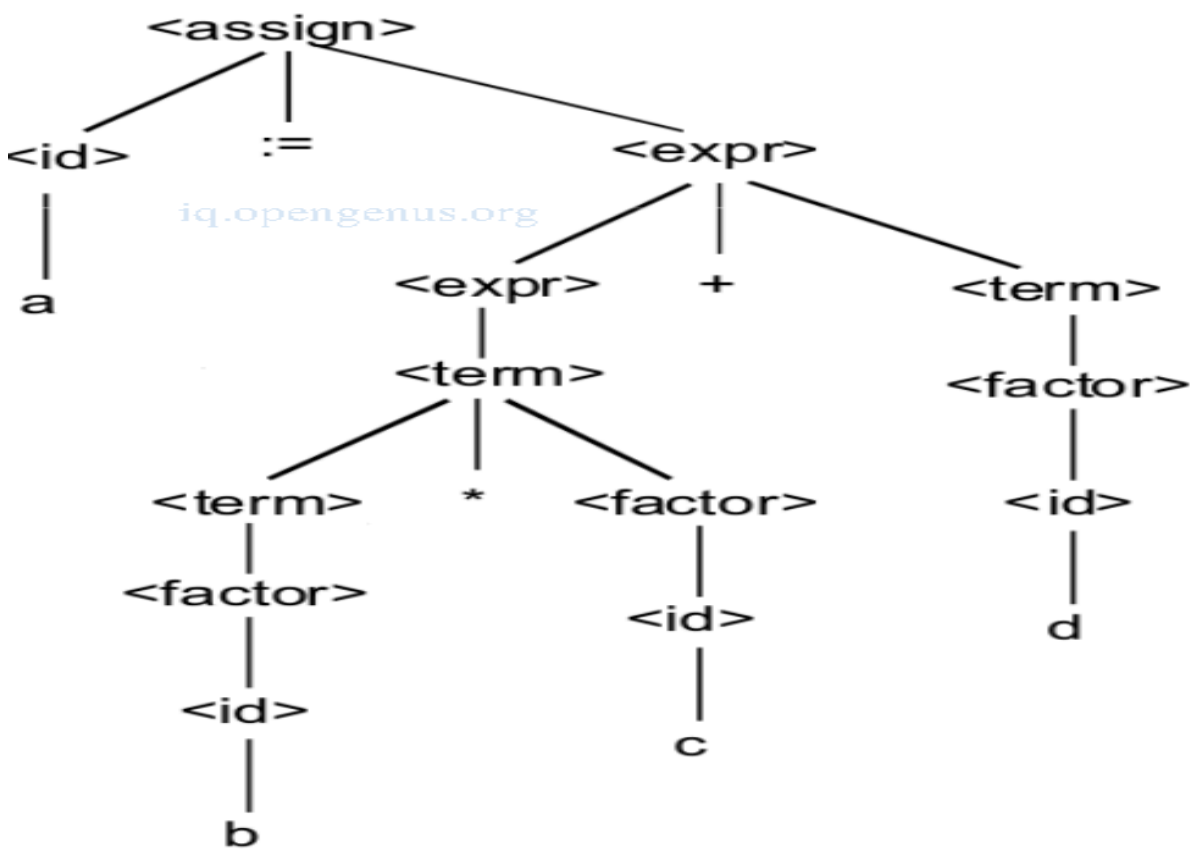
They don't check for string membership in the language of the grammar.

They represent relationships between language constructs and avoid derivations.

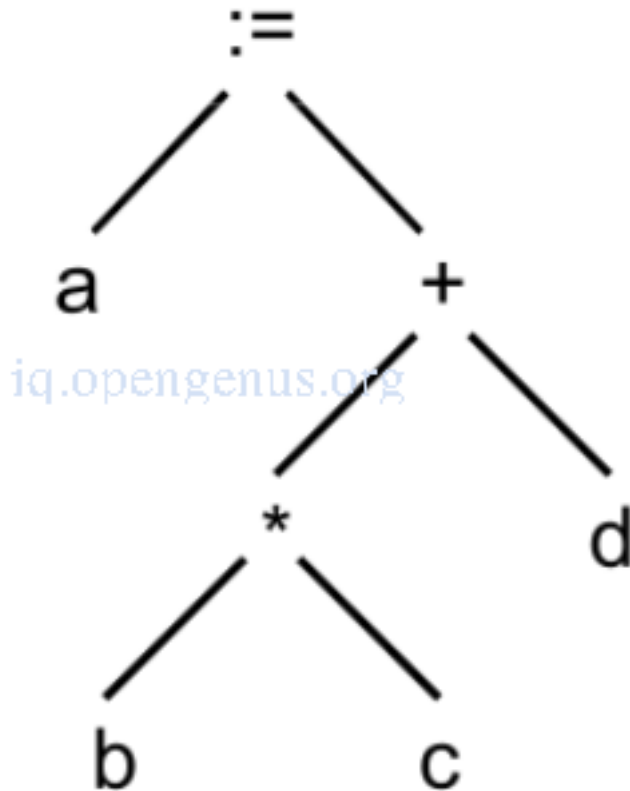
An example

The parse tree and abstract syntax tree for the expression $a := b * c + d$ is.

The parse tree



The abstract syntax tree



Properties of abstract syntax trees.

- Good for optimizations.
- Easier evaluation.
- Easier traversals.
- Pretty printing(unparsing) is possible by in-order traversal.
- Postorder traversal of the tree is possible given a postfix notation.

Implementing Semantic Actions during Recursive Descent parsing.

During this parsing there exist a separate function for each non-terminal in the grammar. The procedures will check the lookahead token against the terminals it expects to find. Recursive descent recursively calls procedures to parse non-terminals it expects to find. At certain points during parsing appropriate semantic actions that are to be performed are implemented.

Roles of this phase.

- Collection of type information and type compatibility checking.
- Type checking.
- Storage of type information collected to a symbol table or an abstract syntax tree.
- In case of a mismatch, type correction is implemented or a semantic error is generated.
- Checking if source language permits operands or not.

CHAPTER 3

SYSTEM ARCHITECTURE AND DESIGN

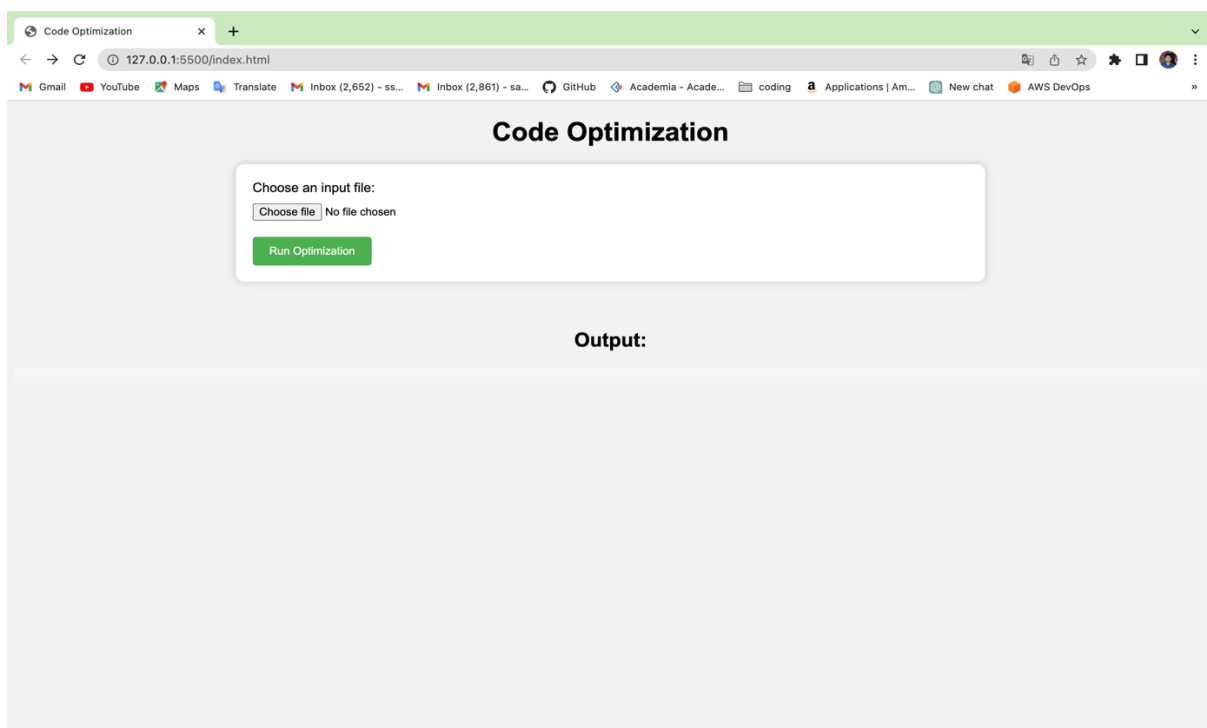
3.1 FRONT-END DESIGN:-

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

1. **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:
2. **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.
3. **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.
4. **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.
5. **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:
6. **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.

7. **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.



3.2 BACK-END DESIGN:-

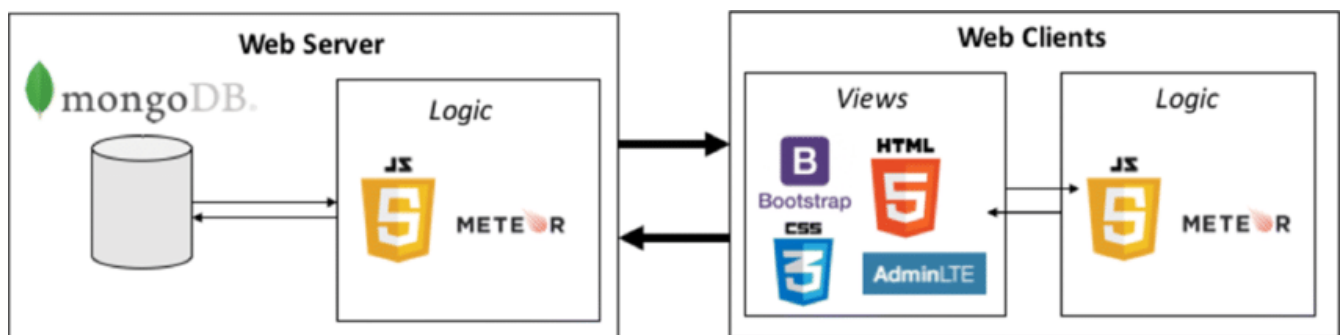
Flask is a popular back-end web framework for Python. It provides a lightweight and flexible approach to building web applications. When using Flask as a back-end framework, it typically consists of the following key components:

1. **Routing:** Flask allows you to define URL routes and associate them with specific functions or methods. These routes determine how the application responds to different URLs or HTTP methods (GET, POST, etc.). By using decorators or URL patterns, you can map routes to corresponding view functions or class methods.
2. **Views:** Views in Flask are Python functions or class methods that handle requests and generate responses. They receive data from requests, process it, and return appropriate responses. Views can render templates, return JSON data, redirect to other URLs, or perform other actions based on the application's requirements.
3. **Templates:** Flask integrates with template engines (such as Jinja2) to separate the presentation logic from the application logic. Templates allow you to dynamically generate HTML pages by incorporating data and logic. Flask provides support for template inheritance, variable substitution, control structures, and other template features.
4. **Forms:** Flask provides utilities for handling HTML forms, including form validation, data retrieval, and rendering. It allows you to define form classes, specify validation rules, and generate HTML form elements. Flask also supports form submission handling, including processing form data and handling validation errors.
5. **Middleware:** Flask allows the use of middleware, which are components that intercept and process requests and responses before they reach the view functions. Middleware can perform tasks such as authentication, logging, error handling, or modifying the request/response objects.
6. **Extensions:** Flask has a rich ecosystem of extensions that provide additional functionality and integrate with various services. These extensions cover areas such as database integration, authentication, session management, caching, API development, and more. Extensions can be easily integrated into Flask applications to enhance their capabilities.

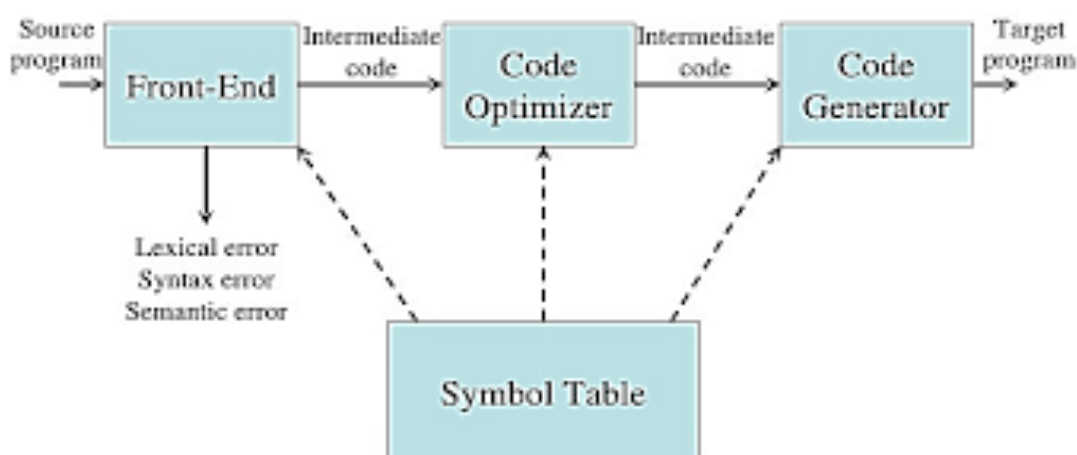
7. **Configuration:** Flask allows you to configure various aspects of the application, such as database connections, debugging options, logging settings, and more. Configuration can be done through environment variables, configuration files, or programmatically in the application code.

These components of Flask form the foundation for developing back-end web applications. They provide the necessary tools and abstractions to handle routing, views, templates, database interactions, form handling, middleware, and configuration. Flask's simplicity and flexibility make it a popular choice for building web applications of varying sizes and complexities.

Backend/Frontend CODE OPTIMIZER ARCHITECTURE DESIGN:-



Component Diagram:-



CHAPTER 4

The requirement to run the script –

To run a semantic analyzer code built using Flask and JavaScript, you will need the following requirements:

1. **A compatible version of Python:** Flask is a web framework for Python, so you will need to have Python installed on your computer. The specific version of Python required may vary depending on the version of Flask and other dependencies used in the code.
2. **Flask and its dependencies:** Flask is a third-party library for Python, and it has several dependencies that must be installed to use it. These dependencies may include Werkzeug, Jinja2, and others. You can use a package manager like pip to install these dependencies.
3. **A web server:** Flask is a web framework, and it requires a web server to run. You can use the built-in development server that comes with Flask, or you can use a production-ready web server like Apache or Nginx.
4. **A browser:** The JavaScript code used in the semantic analyzer will be executed in the user's browser, so you will need a modern web browser like Chrome, Firefox, or Safari to view and interact with the analyzer.
5. **Code editor or IDE:** To edit the Flask and JavaScript code, you will need a code editor or integrated development environment (IDE) that supports Python and JavaScript.
6. **Semantic analysis libraries:** Depending on the specific requirements of your semantic analyzer, you may need to install additional libraries or tools for semantic analysis, such as Natural Language Processing (NLP) libraries or machine learning frameworks.

CHAPTER 5

CODING AND TESTING

CODING:-

1. Main.js:-

```
const fs = require('fs');

// Read input from file
const input = fs.readFileSync('input.txt', 'utf8').trim().split('\n').map(line =>
line.split(','));

// Find and replace common expressions
for (let i = 0; i < input.length; i++) {
  for (let j = i + 1; j < input.length; j++) {
    if (input[i][1] === input[j][1]) {
      for (let d = 0; d < input.length; d++) {

        const right = input[d][1];
        const left = input[d][0];
        const b = Array.from(right).map(() => right.length);

        for (let z = 0; z < b[d]; z++) {
          if (right[z] === input[j][0]) {
            const x = Array.from(right);
            x[z] = input[i][0];
            const l = x.join('');
            input[d][1] = right.replace(input[j][0], input[i][0]);
          }
        }
      }
      input[j][0] = input[i][0];
    }
  }
}

// Write intermediate output to file
let output = input.map(line => line.join(',')).join('\n');
fs.writeFileSync('output.txt', output);

output = fs.readFileSync('output.txt', 'utf8').trim().split('\n').map(line =>
line.split(','));
let i = 0;

let j = 1;
while (j < output.length) {
  if (output[i][1] === output[j][1]) {
    if (output[i][0] === output[j][0]) {
```



```

        output.splice(j, 1);
        i += 2;
        j = i + 1;
    } else {
        i += 1;
    }
} else {
    j += 1;
}
if (j === output.length) {
    i = i + 1;
    j = i + 1;
}
if (i === output.length) {
    j = output.length;
}
}
// Write intermediate output to file
output = output.map(line => line.join(',')).join('\n');
fs.writeFileSync('output.txt', output);
// Remove dead code
output = fs.readFileSync('output.txt', 'utf8').trim().split('\n').map(line =>
line.split(','));
let count = 0;
i = 0;
j = 0;
while (j < output.length && i < output.length) {
    const b = Array.from(output[j][1]).map(() => output[j][1].length);
    for (let z = 0; z < b[0]; z++) {
        if (output[j][1][z] === output[i][0]) {
            count = 1;
        }
    }
    j += 1;
    if (j === output.length) {
        if (count !== 1) {
            output.splice(i, 1);
            i = 0;
            j = 0;
        } else {
            i += 1;
            j = 0;
        }
    }
}
output = output.map(line => line.join('=')).join('\n');
fs.writeFileSync('output.txt', output);
// Print final output to console
console.log('The final optimized code is...');
console.log(output);

```

2. Index.HTML:-(contains the structure and the JS code that analyses the code)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Code Optimization</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1>Code Optimization</h1>
    <script src="main.js"></script>
    <form>
      <label for="inputFile">Choose an input file:</label>
      <input type="file" id="inputFile">
      <br>
      <button type="button" onclick="runOptimization()">Run Optimization</button>
    </form>
    <br>
    <h2>Output:</h2>
    <pre id="output"></pre>
    <script>
      function runOptimization() {
        const inputFile = document.getElementById("inputFile").files[0];
        const reader = new FileReader();
        reader.onload = function(event) {
          const inputText = event.target.result;
          // Run the optimization code here
          // and output the results to the "output" element
          // Example:
          const outputTxt = fetch('output.txt')
            .then(response => response.text())
            .then(text => document.getElementById('output').textContent = text);
          const outputText = inputText.toUpperCase();
          document.getElementById("output").textContent = outputText;
        };
        reader.readAsText(inputFile);
      }
    </script>
  </body>
</html>
```

3. Style.CSS:-

```
body {
  font-family: Arial, sans-serif;
  background-color: #f2f2f2;
}

h1 {
  text-align: center;
  margin-top: 20px;
}

form {
  margin: 20px auto;
  width: 60%;
  padding: 20px;
  background-color: #fff;
  border-radius: 10px;
  box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.2);
}

label {
  display: block;
  margin-bottom: 10px;
}

input[type="file"] {
  margin-bottom: 20px;
}

button[type="button"] {
  background-color: #4CAF50;
  color: #fff;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

button[type="button"]:hover {
  background-color: #3e8e41;
}

h2 {
  margin-top: 20px;
  text-align: center;
}

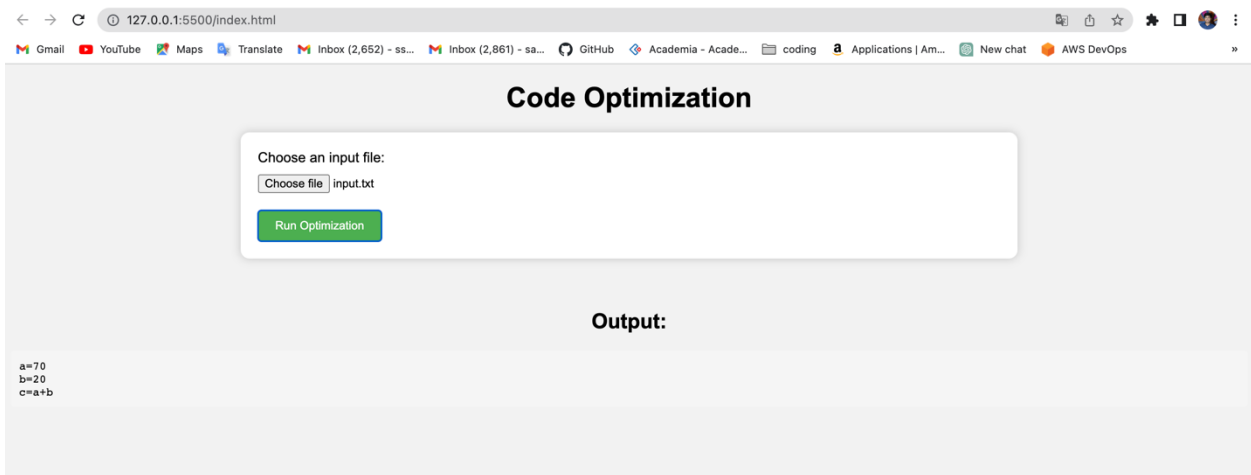
pre {
  background-color: #f5f5f5;
  padding: 10px;
  border-radius: 4px;
  overflow: auto;
}
```

Testing with different inputs

INPUT -1: DEAD CODE ELIMINATION

≡ input.txt

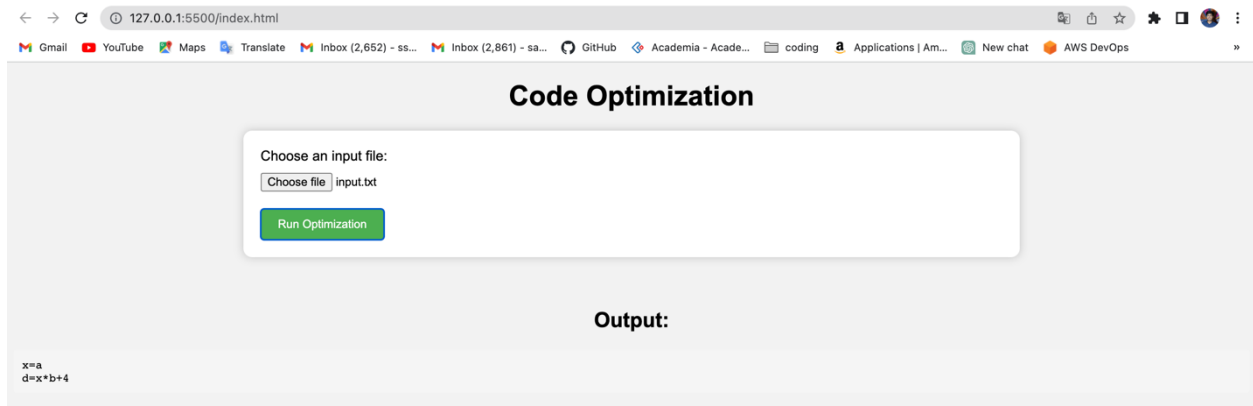
```
1 left, right
2 a, 70
3 b, 20
4 d, 30
5 c, a+b
```



INPUT-2: VARIABLE PROPAGATON

≡ input.txt

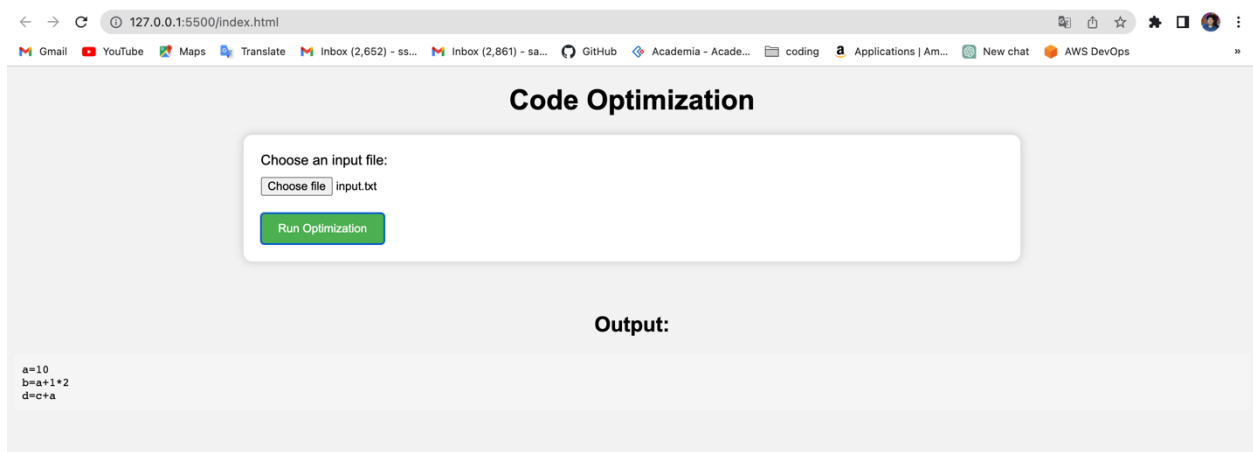
```
1 left, right
2 c, a*b
3 x, a
4 d, x*b+4
5
```



INPUT-3: COMMON SUB-EXPREISSION

≡ input.txt

```
1 left, right
2 a, 10
3 b, a+1*2
4 c, a+1*2
5 d, c+a
6
```

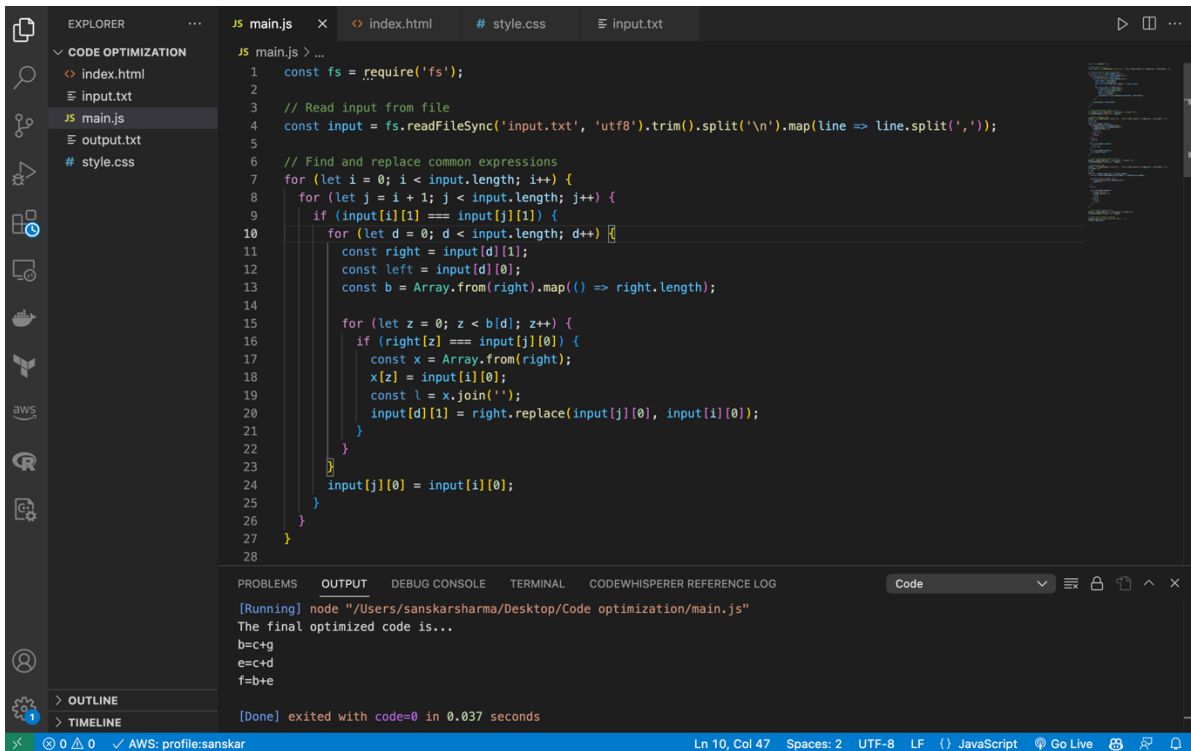


CHAPTER 6

OUTPUT AND RESULTS

1.1 OUTPUT:-

Terminal -



```
1  const fs = require('fs');
2
3  // Read input from file
4  const input = fs.readFileSync('input.txt', 'utf8').trim().split('\n').map(line => line.split(','));
5
6  // Find and replace common expressions
7  for (let i = 0; i < input.length; i++) {
8    for (let j = i + 1; j < input.length; j++) {
9      if (input[i][1] === input[j][1]) {
10         for (let d = 0; d < input.length; d++) {
11           const right = input[d][1];
12           const left = input[d][0];
13           const b = Array.from(right).map(() => right.length);
14
15           for (let z = 0; z < b[d]; z++) {
16             if (right[z] === input[j][0]) {
17               const x = Array.from(right);
18               x[z] = input[i][0];
19               const l = x.join('');
20               input[d][1] = right.replace(input[j][0], input[i][0]);
21             }
22           }
23         }
24         input[j][0] = input[i][0];
25       }
26     }
27   }
28 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL CODEWHISPERER REFERENCE LOG

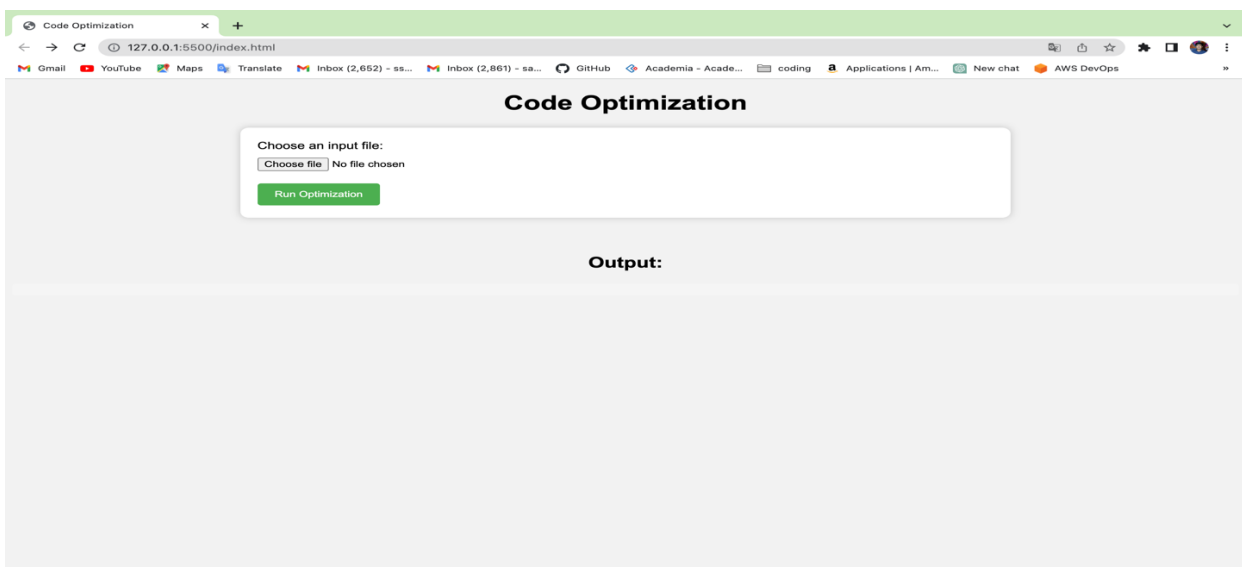
[Running] node "/Users/sanskarsharma/Desktop/Code optimization/main.js"

The final optimized code is...

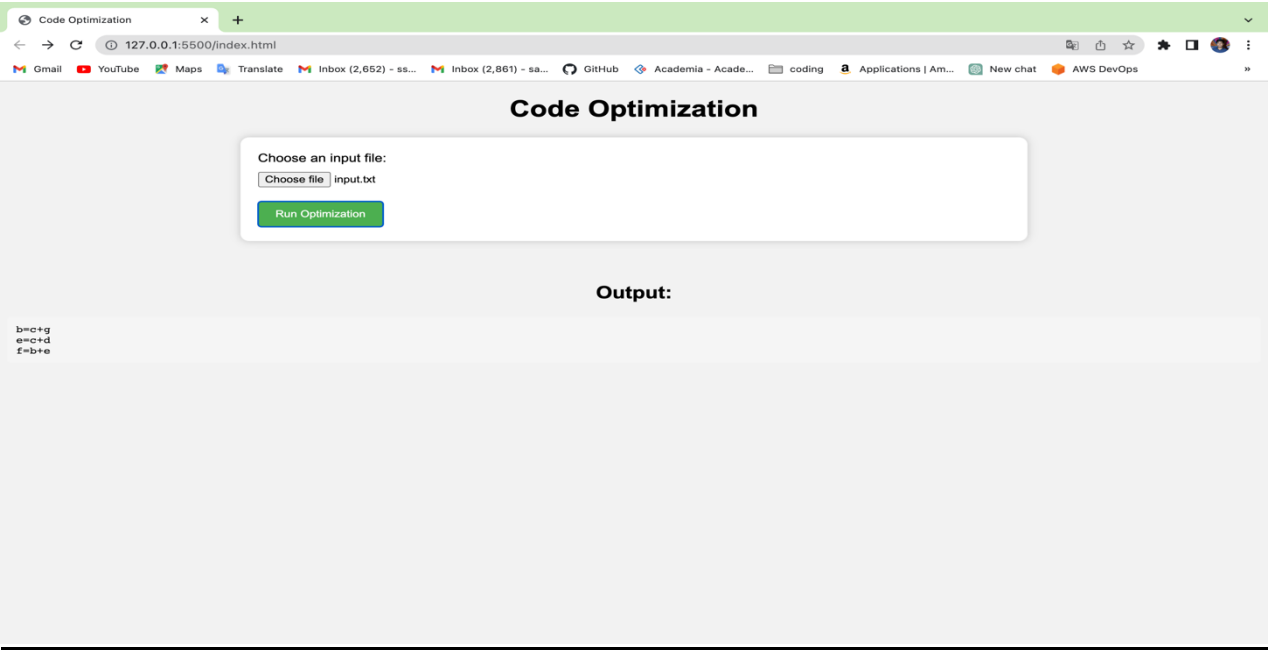
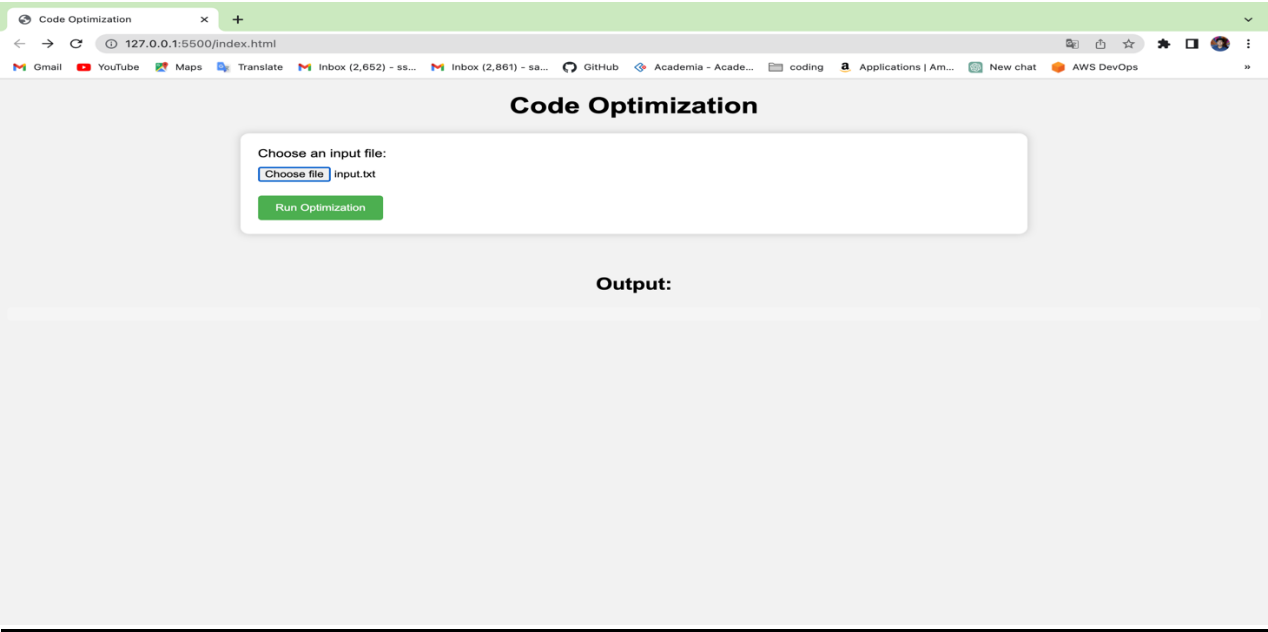
b=c+g
e=c+d
f=b+e

[Done] exited with code=0 in 0.037 seconds

Home Page -



Testing the Optimizer -



1.2 RESULT

The result of the Code Optimizer tool is the optimized code that is generated from the input code. The optimized code is typically faster, more efficient, and more maintainable than the original

code, and can significantly improve the performance of software programs.

The specific results of the Code Optimizer tool depend on the optimization techniques that are applied to the input code. Some common optimization techniques that are used by the Code Optimizer tool include:

- Loop unrolling: This technique involves replacing loops in the code with equivalent code that does not use loops. This can improve the performance of the code by reducing the number of loop iterations.
- Dead code elimination: This technique involves identifying and removing code that is never executed or that has no effect on the output of the program. This can improve the efficiency of the code by reducing the amount of computation that is performed.
- Constant propagation: This technique involves replacing variables with constant values where possible. This can improve the efficiency of the code by reducing the number of memory accesses that are performed.
- Function inlining: This technique involves replacing function calls with the code that is executed by the function. This can improve the performance of the code by reducing the overhead of function calls.

The results of the Code Optimizer tool can be measured using a range of metrics, including execution time, memory usage, and code size. By comparing the results of the optimized code with the original code, developers can determine the effectiveness of the optimization techniques that were applied and make further optimizations as needed.

Overall, the result of the Code Optimizer tool is optimized code that can significantly improve the efficiency and performance of software programs. By automating the code optimization process, developers can save time and effort and focus on other aspects of software development, such as feature development and testing.

CHAPTER 7

7.1 **CONCLUSION:-**

In conclusion, the Code Optimizer project is a powerful tool that can significantly improve the efficiency and performance of software programs. By automating the code optimization process, developers can save time and effort and focus on other aspects of software development. The modular architecture of the tool and its ability to be customized to meet the specific needs of different programming languages make it a valuable tool for developers working on a wide range of software development projects.

The Code Optimizer project uses a range of optimization techniques, including data flow analysis, control flow analysis, and symbolic execution, to identify opportunities for optimization in the input code. The specific optimization techniques that are applied to the input code depend on the optimization settings that are configured by the user.

The Code Optimizer project is implemented using Python and consists of a backend component and a user interface component. The backend component is responsible for analyzing and optimizing the input code, while the user interface component enables developers to interact with the tool and view the optimized code.

Overall, the Code Optimizer project is a valuable tool for developers who want to improve the performance and efficiency of their software programs. By automating the code optimization process, developers can reduce the time and effort required to optimize their code and focus on other aspects of software development.

REFERENCES

1. <https://iq.opengenus.org/semantic-analysis-in-compiler-design/>
2. <http://www.icet.ac.in/Uploads/Downloads/M4.pdf>
3. https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm#:~:text=Semantics,derives%20any%20meaning%20or%20not.