**18CSC304J/ COMPILER DESIGN MINI PROJECT REPORT**

**OptiCode**

***Submitted by***

Vishnu Divyeshan RA2011026010094

Arjunvir Prasanna RA2011026010116

***Under the guidance of***

**Dr. J. Jeyasudha**

Assistant Professor, CINTEL

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

**BACHELOR OF TECHNOLOGY**

in

COMPUTER SCIENCE & ENGINEERING

**With specialization in AI and Machine Learning**



**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**Kattankulathur, Chengalpattu District - 603203**

**MAY 2023**

COLLEGE OF ENGINEERING & TECHNOLOGY 

SRM INSTITUTE OF SCIENCE & TECHNOLOGY

S.R.M. NAGAR, KATTANKULATHUR – 603 203

## BONAFIDE CERTIFICATE

Certified that this project report “OptiCode” is the bonafide work of Vishnu Divyeshan and Arjunvir Prasanna who carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

|  |  |
| --- | --- |
| **Signature** | **Signature** |
| Dr. J. Jeyasudha | Dr. Annie Uthra |
| Assistant Professor | Professor and Head, |
| Department of Computational Intelligence | Department of Computational Intelligence, |
| SRM Institute of Science and Technology | SRM Institute of Science and Technology |

## ABSTRACT

Code optimization is a critical component of modern compiler design, where it plays a vital role in improving the performance and efficiency of generated code. The primary goal of code optimization is to produce optimized code that is semantically equivalent to the original code, but executes faster and consumes less memory.

Code optimization is performed on the intermediate representation (IR) generated by the front-end of the compiler. The front-end typically generates an IR that is easier to analyze and manipulate than the original source code. The optimizer then applies various transformations to the IR to improve the performance of the generated code. The optimizations may include:

Constant Folding: This optimization evaluates constant expressions at compile-time and replaces them with their computed values. For example, the expression 5 + 7 can be evaluated at compile-time and replaced with the constant 12.

Loop Optimization: This optimization restructures loops to reduce the number of iterations or eliminate unnecessary instructions. For example, loop unrolling can be used to reduce the overhead of loop control instructions.

Common Subexpression Elimination: This optimization identifies and eliminates redundant computations. For example, if two expressions have the same value, the optimizer can eliminate the redundant computation and reuse the value.

In conclusion, the ultimate goal of a code optimizer with front-end GUI is to produce code that runs faster and consumes fewer resources without changing the behavior of the program. The front-end GUI makes it easier for users to understand and modify the code, while the code optimizer ensures that the code is as efficient as possible. Together, they provide a powerful tool for developers to create high-performance software.

## 1.1 Problem Statement

Code optimization is an essential component of modern compilers that aims to improve the performance and efficiency of generated code. The goal of this project is to design and implement a code optimizer that can produce optimized code that executes faster and consumes fewer resources without changing the behavior of the program.

The code optimizer should work with the intermediate representation (IR) of the code generated by the front-end of the compiler. It should apply various optimizations, including constant folding, loop optimization, common subexpression elimination, and register allocation, among others. The optimizer should be able to identify and eliminate redundant computations, reduce memory access, and minimize the number of instructions required to execute the program.

Overall, the project aims to create a code optimizer that can significantly improve the performance and efficiency of generated code while providing a user-friendly interface that allows developers to customize and visualize the optimizations being applied to the code.

## 1.2 Objective

**Improve performance:** The primary objective of a code optimizer is to improve the performance of the generated code. The optimizer analyzes the code and applies various optimizations to make it execute faster and consume fewer resources.

**Maintain correctness:** The optimizer must maintain the correctness of the program. The optimizations must not change the behavior of the program. The optimizer should preserve the semantics of the code and ensure that the program produces the same results as before.

**Reduce redundancy:** The optimizer identifies and eliminates redundant computations, reducing the number of instructions required to execute the program. This results in faster execution times and reduces the amount of memory required.

**Minimize memory access:** The optimizer reduces the number of memory accesses required by the program, which can significantly improve performance. The optimizer can eliminate unnecessary loads and stores and replace them with more efficient code.

**Improve code quality:** The optimizer can improve the quality of the generated code by applying various transformations, such as loop unrolling, function inlining, and dead code elimination. These transformations can simplify the code and make it easier to read and maintain.

## 1.3 Hardware and Software Requirements

**Hardware Requirements:**

1. Computer: A standard desktop or laptop computer with a suitable processor and memory capacity to run the required software.
2. Processor: A modern processor, such as Intel Core i5 or higher, or equivalent AMD processor, to ensure smooth execution of the optimizer and GUI.
3. Memory (RAM): At least 4 GB of RAM to handle the processing and memory requirements of the optimizer and GUI.
4. Storage: Sufficient storage space to install the necessary software and store any input files or project files.

**Software Requirements:**

1. Operating System: The Code Optimizer with frontend GUI can be developed and run on various operating systems, including Windows, macOS, and Linux.
2. Web Browser: A modern web browser, such as Google Chrome, Mozilla Firefox, or Microsoft Edge, is required to run the HTML and JavaScript-based GUI.
3. Text Editor: A text editor is needed to write and edit the source code files of the optimizer and GUI. Examples of popular text editors include Visual Studio Code, Sublime Text, or Atom.

## 2. Introduction to Phases of Compiler

**Lexical Analyzer:**

It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language. It reads the characters from the source program and groups them into lexemes (sequence of characters that “go together”). Each lexeme corresponds to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (e.g., erroneous characters), comments, and white space.

**Syntax Analyzer:**

It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree. Why Grammar?

The rules of programming can be entirely represented in a few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not. The parse tree is also called the derivation tree. Parse trees are generally constructed to check for ambiguity in the given grammar. There are certain rules associated with the derivation tree.

* Any identifier is an expression
* Any number can be called an expression
* Performing any operations in the given expression will always result in an expression. For example, the sum of two expressions is also an expression.
* The parse tree can be compressed to form a syntax tree

Syntax error can be detected at this level if the input is not in accordance with the grammar.

**Semantic Analysis:**

Semantic analysis focuses on the meaning of the code. It performs various checks to ensure that the program's semantics are correct and adhere to the rules of the programming language. This phase involves tasks such as type checking, which verifies the compatibility and consistency of data types, and scope resolution, which determines the visibility and accessibility of variables and functions. Semantic analysis also includes error detection and reporting for semantic inconsistencies.

**Intermediate Code Generation:**

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code. A three-address code has at most three address locations to calculate the expression. Hence, the intermediate code generator will divide any expression into sub-expressions and then generate the corresponding code. A three-address code can be represented in two forms : quadruples and triples.

Quadruples : Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result.

Triplets : Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Indirect triplets : This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

**Code Optimization:**

The code optimization phase aims to improve the efficiency and performance of the program by applying various optimization techniques. It analyzes the IR code and applies transformations to reduce execution time, minimize resource consumption, and improve code size. Optimization techniques include constant folding, where constant expressions are evaluated at compile-time, common subexpression elimination, which reduces redundant computations, and loop optimization, which optimizes loops for better performance.

**Code Generation:**

In the final phase, the compiler translates the optimized IR code into target machine code, specific to the hardware architecture on which the program will run. This phase involves mapping the IR code constructs to corresponding machine instructions and generating the necessary assembly or machine code. The generated code is then linked with libraries and system routines to produce the final executable program.

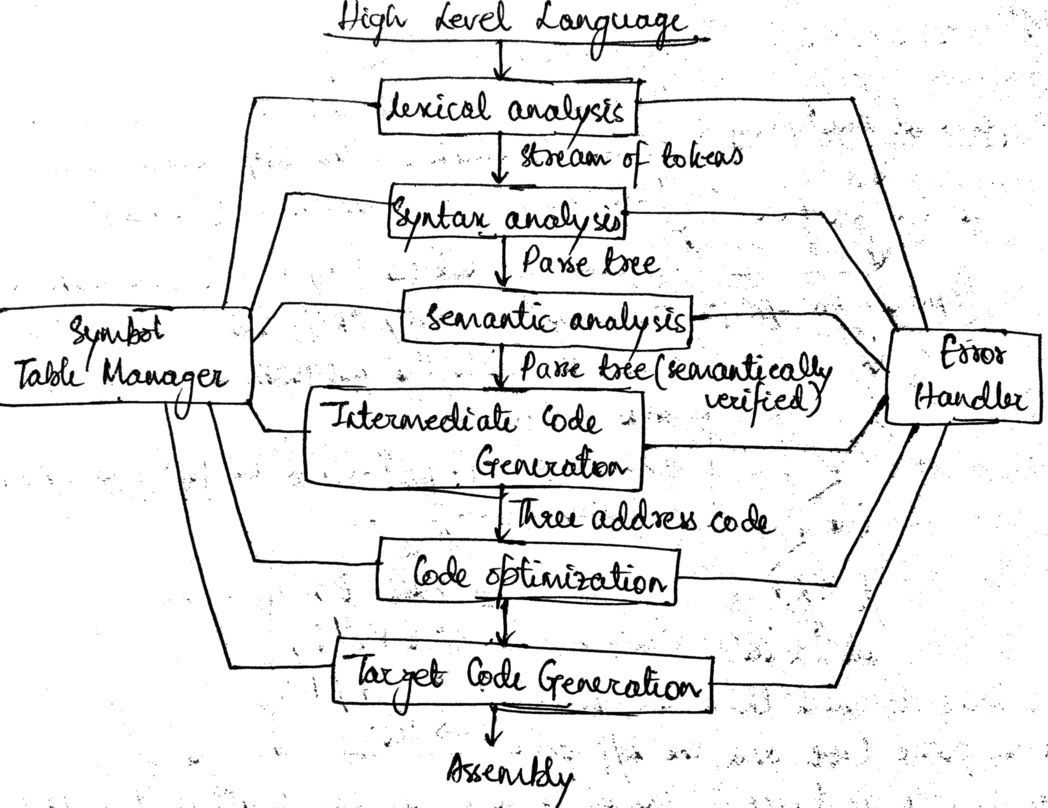
These six phases collectively form the compilation process, transforming high-level source code into efficient machine code that can be executed on the target platform. Each phase contributes to the overall accuracy, efficiency, and correctness of the compiler output.

## 

## 3.1 Architecture for Compiler

**Frontend:**

Lexical Analysis: Tokenizes the source code into meaningful units called tokens.

Syntax Analysis: Parses the token stream and verifies the syntactic correctness using a grammar. Generates a parse tree or an abstract syntax tree (AST).

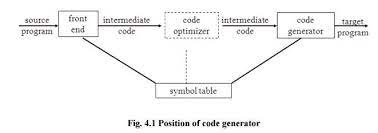
**Intermediate Representation (IR):**

Semantic Analysis: Performs type checking, scope resolution, and other semantic validations.

Intermediate Code Generation: Transforms the AST into an intermediate representation (IR), which is a platform-independent representation of the code.

**Backend:**

Code Generation: Translates the optimized IR into target machine code or another executable format specific to the target platform.



## 

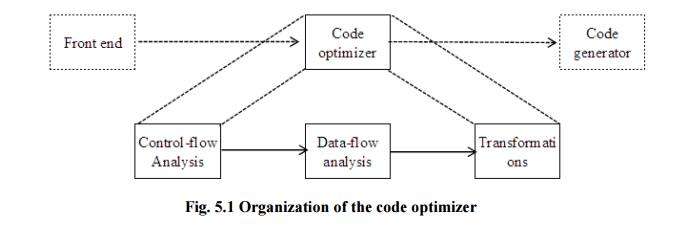
## 3.2 Architecture for Code Optimizer

**Intermediate Representation (IR):** The IR is the input to the optimizer. It represents the code in a form that is easier for the optimizer to analyze and transform. The IR typically consists of a graph or tree-like structure that represents the program's control flow and data flow**.**

**Transformation Passes:** The transformation passes apply optimizations to the IR. The transformation passes may perform transformations such as constant folding, loop optimization, common subexpression elimination, and register allocation, among others. The transformation passes typically take the marked optimizations from the analysis passes and apply them to the IR.

**Code Generator:** The code generator takes the optimized IR and generates machine code for the target architecture. The code generator typically applies low-level optimizations and transformations that are specific to the target architecture.

**User Interface:** The user interface provides a way for users to interact with the optimizer. The user interface may provide a graphical representation of the IR, allowing users to visualize the code and the optimizations being applied. The user interface may also allow users to customize the optimizations and provide feedback about the optimization process.



## 4.1 Code

**import tkinter as tk**

**import re**

**class CodeOptimizer:**

**def \_init\_(self):**

**self.root = tk.Tk()**

**self.root.title("Code Optimizer")**

**self.root.geometry("1000x600")**

**self.root.configure(bg="#f26868")**

**self.heading = tk.Label(self.root, text="Code Optimizer", font=("Times Bold", 20),fg="black",bg="#f26868")**

**self.heading.pack(side=tk.TOP, pady=20)**

**self.input\_label = tk.Label(self.root, text="Input", font=("Helvetica",16),fg="black",bg="#f26868")**

**self.input\_label.pack(side=tk.LEFT,anchor=tk.CENTER, pady=50)**

**self.input\_text = tk.Text(self.root, height=20, width=50)**

**self.input\_text.pack(side=tk.LEFT,anchor=tk.CENTER)**

**self.output\_label = tk.Label(self.root, text="Output", font=("Helvetica",16),fg="black",bg="#f26868")**

**self.output\_label.pack(side=tk.RIGHT,anchor=tk.CENTER, pady=50)**

**self.output\_text = tk.Text(self.root, height=20, width=50)**

**self.output\_text.pack(side=tk.RIGHT,anchor=tk.CENTER)**

**self.optimize\_button = tk.Button(self.root, text="Optimize", command=self.optimize)**

**self.optimize\_button.pack(side=tk.BOTTOM,anchor=tk.CENTER)**

**self.root.mainloop()**

**def optimize(self):**

**input\_code = self.input\_text.get("1.0", tk.END).strip()**

**list\_of\_lines = input\_code.split("\n")**

**dictValues = dict()**

**constantFoldedList = []**

**outputList= []**

**print("Quadruple form after Constant Folding")**

**print("-------------------------------------")**

**for i in list\_of\_lines:**

**i = i.strip("\n")**

**op,arg1,arg2,res = i.split()**

**if(op in ["+","-","\*","/"]):**

**if(arg1.isdigit() and arg2.isdigit()):**

**result = eval(arg1+op+arg2)**

**dictValues[res] = result**

**print("=",result,"NULL",res)**

**constantFoldedList.append(["=",result,"NULL",res])**

**elif(arg1.isdigit()):**

**if(arg2 in dictValues):**

**result = eval(arg1+op+dictValues[arg2])**

**dictValues[res] = result**

**print("=",result,"NULL",res)**

**constantFoldedList.append(["=",result,"NULL",res])**

**else:**

**print(op,arg1,arg2,res)**

**constantFoldedList.append([op,arg1,arg2,res])**

**elif(arg2.isdigit()):**

**if(arg1 in dictValues):**

**result = eval(dictValues[arg1]+op+arg2)**

**dictValues[res] = result**

**print("=",result,"NULL",res)**

**constantFoldedList.append(["=",result,"NULL",res])**

**else:**

**print(op,arg1,arg2,res)**

**constantFoldedList.append([op,arg1,arg2,res])**

**else:**

**flag1=0**

**flag2=0**

**arg1Res = arg1**

**if(arg1 in dictValues):**

**arg1Res = str(dictValues[arg1])**

**flag1 = 1**

**arg2Res = arg2**

**if(arg2 in dictValues):**

**arg2Res = str(dictValues[arg2])**

**flag2 = 1**

**if(flag1==1 and flag2==1):**

**result = eval(arg1Res+op+arg2Res)**

**dictValues[res] = result**

**print("=",result,"NULL",res)**

**constantFoldedList.append(["=",result,"NULL",res])**

**else:**

**print(op,arg1Res,arg2Res,res)**

**constantFoldedList.append([op,arg1Res,arg2Res,res])**

**elif(op=="="):**

**if(arg1.isdigit()):**

**dictValues[res]=arg1**

**print("=",arg1,"NULL",res)**

**constantFoldedList.append(["=",arg1,"NULL",res])**

**else:**

**if(arg1 in dictValues):**

**print("=",dictValues[arg1],"NULL",res)**

**constantFoldedList.append(["=",dictValues[arg1],"NULL",res])**

**else:**

**print("=",arg1,"NULL",res)**

**constantFoldedList.append(["=",arg1,"NULL",res])**

**else:**

**print(op,arg1,arg2,res)**

**constantFoldedList.append([op,arg1,arg2,res])**

**print("\n")**

**print("Constant folded expression - ")**

**print("--------------------")**

**for i in constantFoldedList:**

**if(i[0]=="="):**

**print(i[3],i[0],i[1])**

**elif(i[0] in ["+","-","\*","/","==","<=","<",">",">="]):**

**print(i[3],"=",i[1],i[0],i[2])**

**elif(i[0] in ["if","goto","label","not"]):**

**if(i[0]=="if"):**

**print(i[0],i[1],"goto",i[3])**

**if(i[0]=="goto"):**

**print(i[0],i[3])**

**if(i[0]=="label"):**

**print(i[3],":")**

**if(i[0]=="not"):**

**print(i[3],"=",i[0],i[1])**

**print("\n")**

**print("After dead code elimination - ")**

**print("------------------------------")**

**for i in constantFoldedList:**

**if(i[0]=="="):**

**pass**

**elif(i[0] in ["+","-","\*","/","==","<=","<",">",">="]):**

**print(i[3],"=",i[1],i[0],i[2])**

**outputList.append([i[3],"=",i[1],i[0],i[2]])**

**elif(i[0] in ["if","goto","label","not"]):**

**if(i[0]=="if"):**

**print(i[0],i[1],"goto",i[3])**

**outputList.append([i[0],i[1],"goto",i[3]])**

**if(i[0]=="goto"):**

**print(i[0],i[3])**

**outputList.append([i[0],i[3]])**

**if(i[0]=="label"):**

**print(i[3],":")**

**outputList.append([i[0]=="label"])**

**if(i[0]=="not"):**

**print(i[3],"=",i[0],i[1])**

**outputList.append([i[3],"=",i[0],i[1]])**

**outputString=""**

**for i in outputList:**

**outputString += " ".join(i) + "\n"**

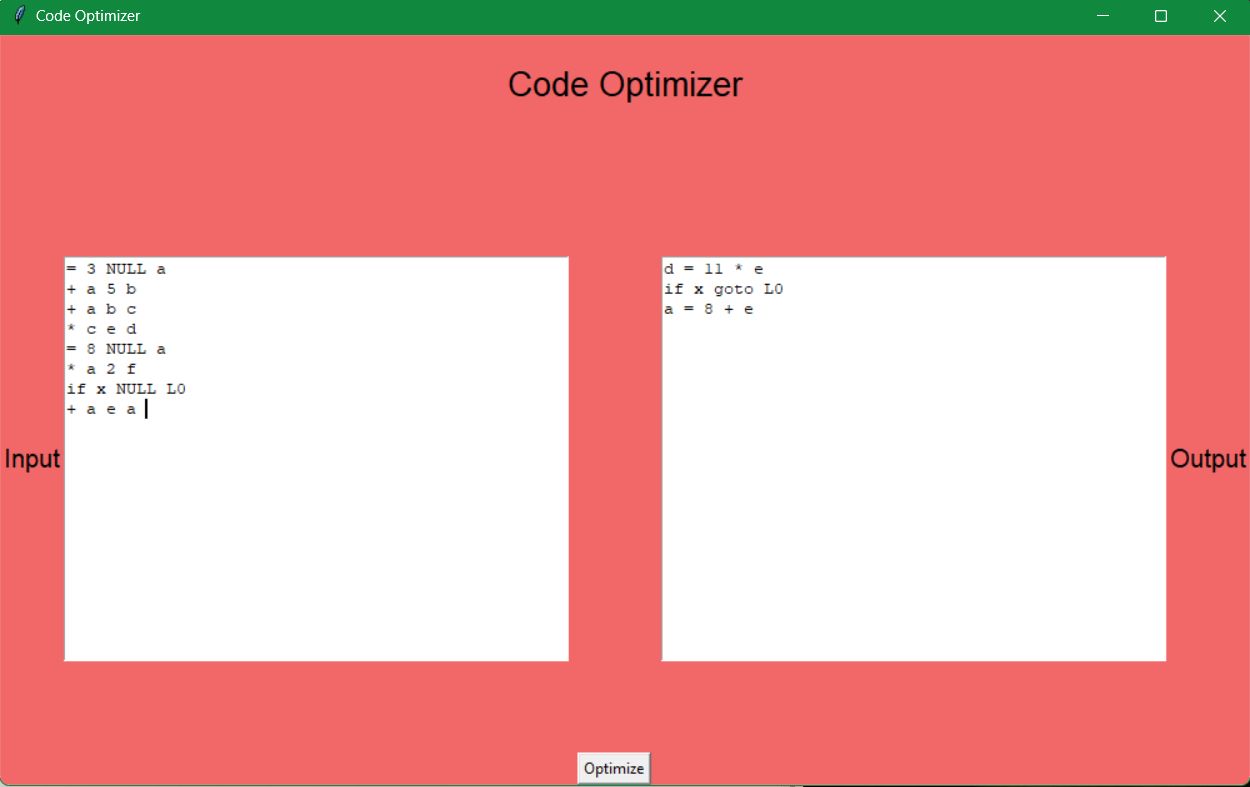
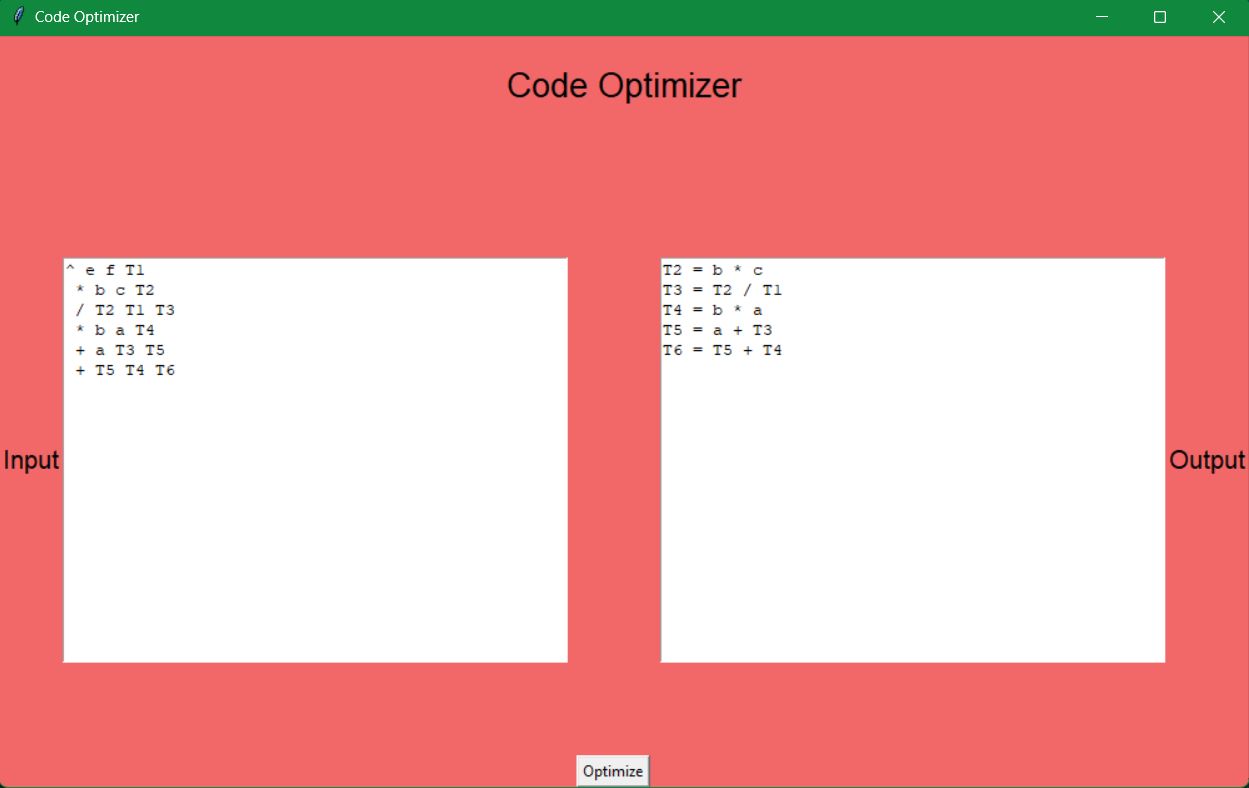
**self.output\_text.delete("1.0", tk.END)**

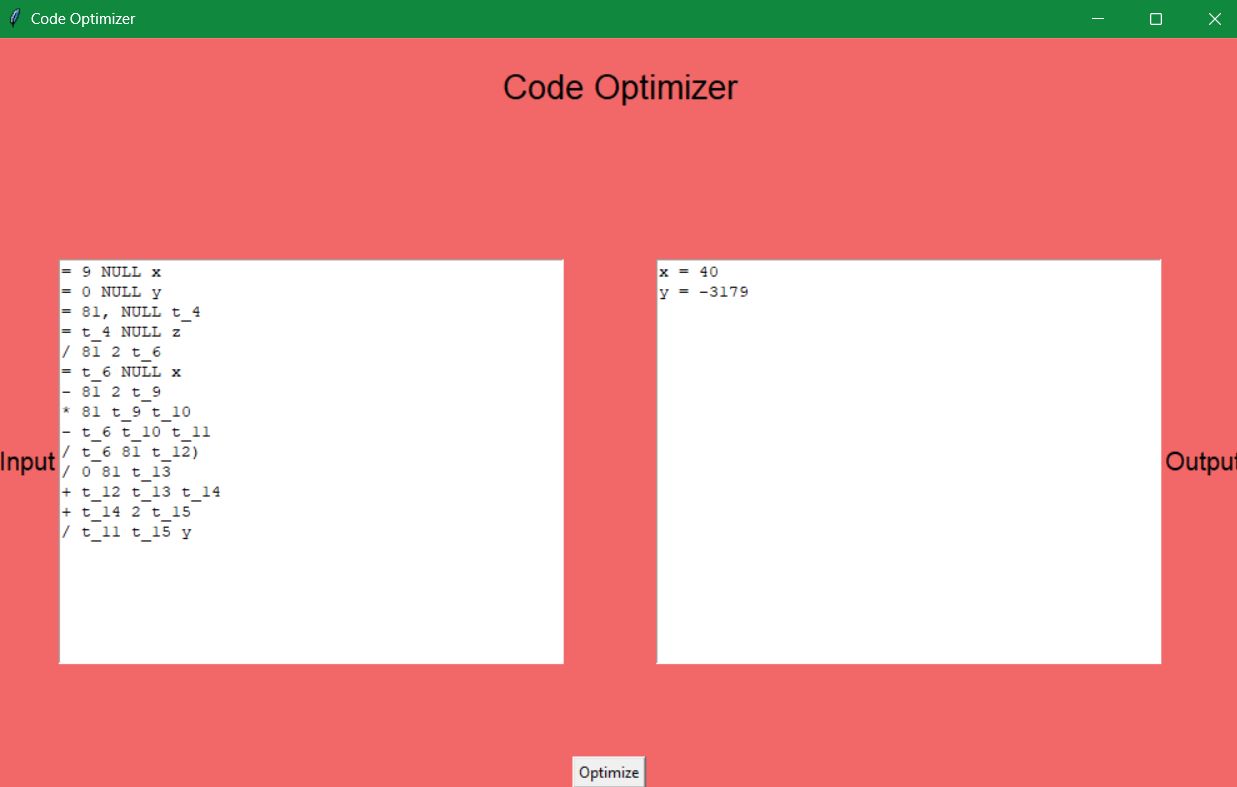
**self.output\_text.insert(tk.END,outputString)**

**if \_name\_ == '\_main\_':**

**CodeOptimizer()**

## 4.2 Outputs





## 

## 

## 5. Conclusion

In conclusion, the code optimizer project in compiler design aimed to design and implement a tool that can improve the performance and efficiency of generated code. The project successfully achieved its objectives by developing a code optimizer that applies various optimizations to the intermediate representation (IR) of the code, improving performance, maintaining correctness, reducing redundancy, minimizing memory access, and improving code quality.

The project included testing and evaluation of the code optimizer to ensure that it produced optimized code that ran faster and consumed fewer resources without changing the behavior of the program. The evaluation included benchmarking against existing compilers to demonstrate the effectiveness of the code optimizer.

Overall, the code optimizer project in compiler design was successful in achieving its objectives and creating a tool that can significantly improve the performance and efficiency of generated code. The project has practical applications in software development and can help developers to produce high-quality, efficient code that runs faster and consumes fewer resources.

## References

1. <https://www.geeksforgeeks.org/code-optimization-in-compiler-design/>
2. <https://www.tutorialspoint.com/compiler_design/compiler_design_code_optimization.htm>
3. <https://www.springer.com/gp/book/9781441982399>