# THE COMPILER

***Submitted by***

# Nitin Manoj [RA2011026010083]

# Aravind Krishnan R[RA2011026010077]

***Under the Guidance of***

# Dr. J. Jeyasudha

**Assistant Professor, Department of Computational Intelligence**

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

# BACHELORS OF TECHNOLOGY

**in**

# COMPUTER SCIENCE ENGINEERING

**with specialization in Artificial Intelligence & Machine Learning**



# SCHOOL OF COMPUTING

**COLLEGE OF ENGINEERING AND TECHNOLOGY SRM INSTITUTE OF SCIENCE AND TECHNOLOGY KATTANKULATHUR - 603203**

**May 2023**



SRM INSTITUTION OF SCIENCE AND TECHNOLOGY KATTANKULATHUR-603203

# BONAFIDE CERTIFICATE

Certified that **18CSC304J – COMPILER DESIGN** project report titled **“THE COMPLIER**” is the bonafide work of **NITIN MANOJ [RA2011026010083] and ARAVIND KRISHNAN R [RA2011026010077]** who carried out project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not perform any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

|  |  |
| --- | --- |
| **SIGNATURE**  Faculty In-Charge **Dr. J. Jeyasudha** Assistant Professor  Department of Computational Intelligence  SRM Institute of Science and Technology Kattankulathur Campus, Chennai | **SIGNATURE**  **HEAD OF THE DEPARTMENT**  **Dr. R Annie Uthra**  Professor and HOD,  Department of Computational Intelligence,  SRM Institute of Science and Technology Kattankulathur Campus, Chennai |

**TABLE OF CONTENTS**

**Chapter No. Title Page No.**

|  |  |
| --- | --- |
| **1 CHAPTER 1 - INTRODUCTION** | **4** |
| 1.1 Introduction | 4 |
| 1.2 Problem statement | 5 |
| * 1. Objectives   2. Hardware & Software Specifications | 6  7 |
| **2 CHAPTER 2 - ANATOMY OF A COMPILER** | **8** |
|  | 8 |
| * 1. Intermediate Code Generation   2. Three address code | 9  10 |
| **3 CHAPTER 3 – ARCHITECTURE AND COMPONENTS** | **11** |
| 3.1 Architecture Diagram | 11 |
| 3.2 Component Diagrams | 12 |
| * + 1. Intermediate Code Generation     2. Quadruple     3. Triple | 12  14  15  16 |
| **4 CHAPTER 4 – CODING AND TESTING** | **17** |
|  | 17 |
| 4.1 Intermediate Code Generation | 20 |
| **5 CHAPTER 5 - RESULT** | **21** |
| **6 CHAPTER 6 - CONCLUSIONS** | **22** |

# CHAPTER 1

* 1. **INTRODUCTION**

The development of a website that allows developers to input their code and see the output of what the compiler would produce is an excellent tool for developers. As software development becomes increasingly complex, the ability to test code and identify potential errors before deploying it is critical. This tool provides developers with a convenient way to quickly and efficiently test their code without the need to install and configure a complete development environment.

Consider a scenario where a developer is working on a new feature for an application. The feature requires the use of a complex algorithm that the developer has not worked with before. The developer writes the code and attempts to run it, but immediately receives an error message from the compiler. Without the ability to test the code in isolation, the developer must spend significant time trying to isolate the error and determine how to fix it.

However, with the use of the tool we have developed, the developer can simply input the code into the website and select the appropriate part of the compiler to test it. In this case, the developer could select the lexical analyzer to identify and correct any syntax errors. The tool quickly identifies the error and provides a clear message to the developer on how to fix it. With the error corrected, the developer can then run the code again and see the output of what the compiler would produce. This tool saves the developer a significant amount of time and effort in the debugging process.

Additionally, this tool is also useful for teaching programming. Beginners can use the tool to learn the basics of programming without the need to install and configure a development environment. The tool provides an easy-to-use interface that allows users to write code, see the output of what the compiler would produce, and make changes as needed. The feedback provided by the tool is immediate and clear, which helps beginners quickly identify and correct errors.

# PROBLEM STATEMENT

As a software developer, we might have encountered situations where you want to test your code against different compilers, or we might have to compile your code on different platforms. But it can be time-consuming and challenging to set up different compilers and platforms to compile your code manually. This is where the tool you have developed comes in handy. The tool allows developers to input their code and see the output of what the compiler would produce without worrying about installing and configuring different compilers and platforms. With your tool, developers can quickly test their code against different compilers and platforms without leaving their development environment. This tool can also be beneficial for developers who are just starting with programming, as they can see how their code is being compiled and understand the different stages of the compilation process, such as lexical analysis and intermediate code generation. Moreover, the tool can help developers to identify and fix errors in their code during the development phase, making it easier for them to deliver bug-free code. Hence, the tool can save developers a lot of time and effort by providing them with a convenient and efficient way to test their code against different compilers and platforms, and help them deliver high-quality code.

# OBJECTIVES

* Develop a working model which will generate three address code for any given expression.
* The above code is developed using python programming language.
* The model should be seen implementing front end.
* GUI should be used for front end.
* tkinter library functions needs to be used.

# HARDWARE AND SOFTWARE SPECIFICATIONS

* + 1. **HARDWARE REQUIREMENTS**
* A server or cloud infrastructure to host the website and the backend logic.
* Sufficient RAM and CPU power to handle multiple user requests simultaneously.
* Sufficient disk space to store the code files and other resources.

# SOFTWARE REQUIREMENTS

* Operating system: Linux or compatible OS.
* Python interpreter installed on the server.
* Tkinter module installed to run the GUI.

# CHAPTER 2 - ANATOMY OF A COMPILER

## 2.1 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.

## 2.2 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. 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

## 2.3 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.

## 2.4 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.

## 2.5 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.

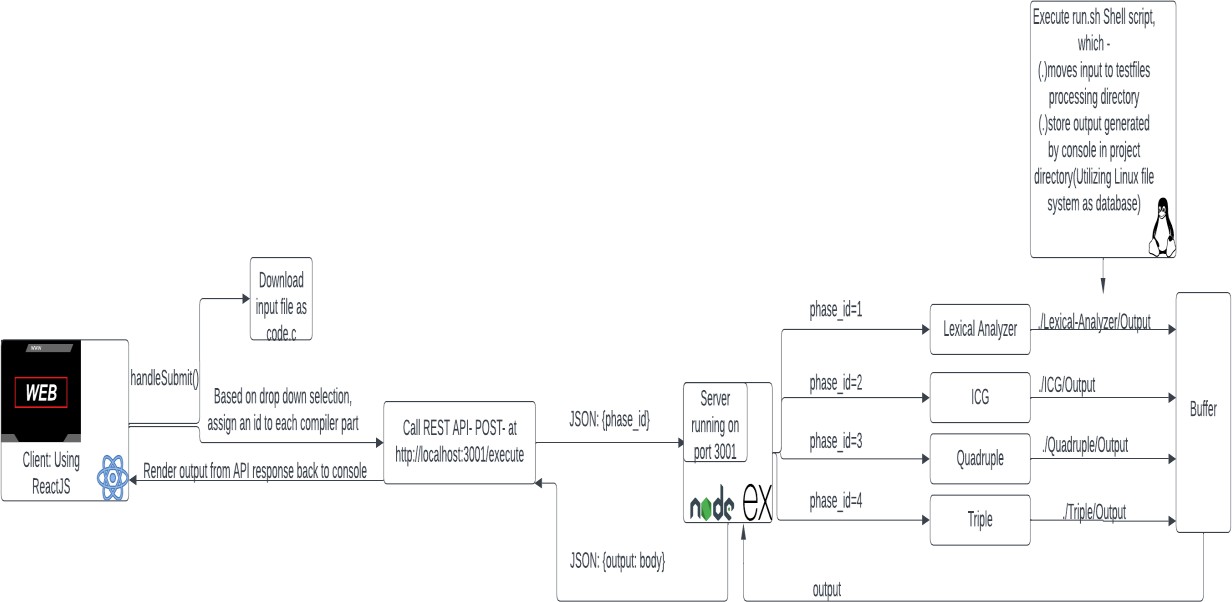
## 2.6 CODE GENERATION

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.

# CHAPTER 3 ARCHITECTURE AND COMPONENTS

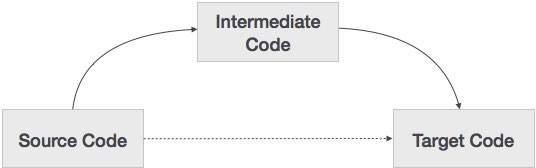
* 1. **ARCHITECTURE DIAGRAM**



The project consists of several components that work together to achieve its goal. The Front-end UI is responsible for presenting the user interface to the end-user, and it is built using NodeJS, which is a popular JavaScript library for building user interfaces. The FLASK serves as the communication interface between the front-end UI and the back-end logic. On the other hand, the back-end logic consists of several components, including the Lexical Analyzer and the Intermediate Code Generator (ICG). The Lexical Analyzer is responsible for analysing the input source code and generating a stream of tokens, while the ICG is responsible for generating an intermediate code representation of the input source code. Three Address code data structures are used to represent the intermediate code generated by the ICG, and they are also implemented as a part of the back- end logic. Overall, the project architecture involves the front-end UI, the FLASK, and the back-end logic components, including the Lexical Analyzer, ICG, and data structures. Each component has a specific role and works together to achieve the project's objective.

# COMPONENTS DIAGRAMS

# INTERMEDIATE CODE GENERATION

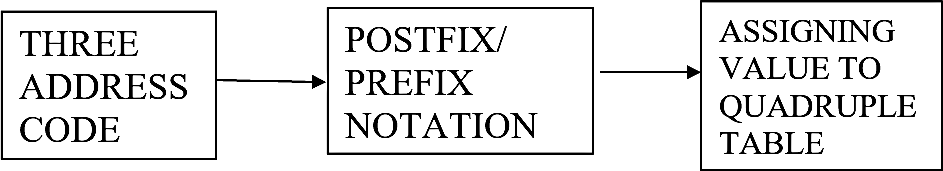


A component diagram is a type of UML diagram that depicts the system's components and their relationships. In the context of intermediate code generation in a compiler, a component diagram can be used to illustrate the flow of information between the different components of the system, namely the source code, intermediate code, and target code.

The source code is the original program written in a high-level language that the compiler receives as input. The intermediate code is an intermediate representation of the program that is generated during the compilation process. The target code is the final output of the compiler, which is often in the form of machine code or assembly language.

In the component diagram, the source code component would be depicted as the input to the system, with an arrow pointing towards the intermediate code component. The intermediate code component would be the central component of the system, as it represents the intermediate representation of the program that is generated during the compilation process. It would have arrows pointing towards both the source code component and the target code component. This indicates that the intermediate code is generated from the source code and is used to generate the target code.

# QUADRUPLE

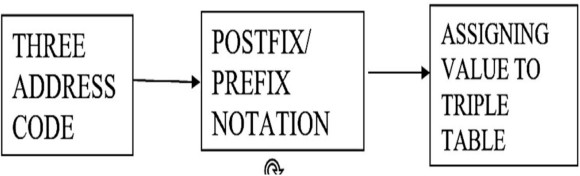


Three-address code would be shown as the system's input on the component diagram, with arrows pointing in the direction of the postfix/prefix expression component and the quadruple component. The three-address code's arithmetic and logical expressions would be translated into postfix/prefix notation by the postfix/prefix expression component. The postfix/prefix expressions would be used to create the quadruples by the quadruple component.

Each sub-component of the quadruple production process, such as parsing the postfix/prefix expressions, constructing the quadruples, and optimising the quadruples, is in charge of a specific step in the quadruple generation process.

The output of the system would be shown as the quadruple component, with an arrow pointing in the direction of the three-address code component. This illustrates how the quadruples are produced.

# TRIPLE



The operation code, the source operand, and the destination operand would each be shown as a separate component in a component diagram for a triple, joined by lines that show their relationships. A line linking the operation code component to the source operand component, for instance, would show that the operation code defines the kind of operation to be carried out on the source operand.

A component diagram for a triple might additionally include additional elements that are utilised across the entire programme, such as variables, constants, and control flow structures, in addition to the three components of the triple. Lines connecting these elements would also show how they

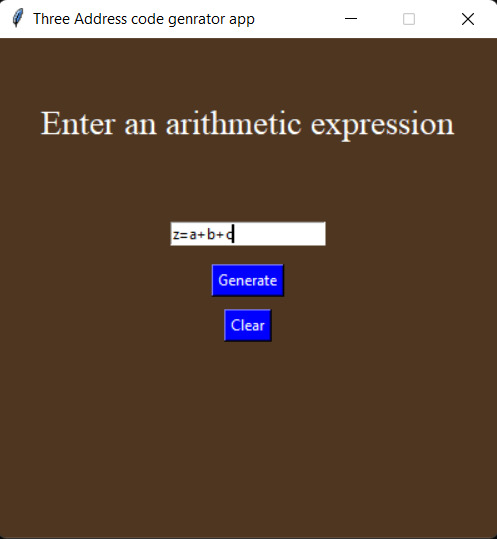
Relate to one anothe

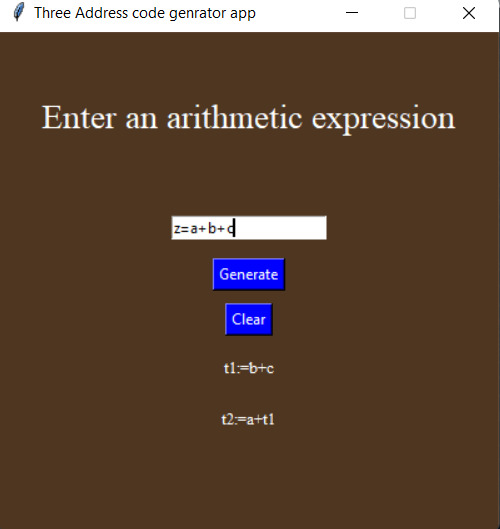
# CHAPTER 4 CODING AND TESTING

# INTERMEDIATE CODE GENERATION

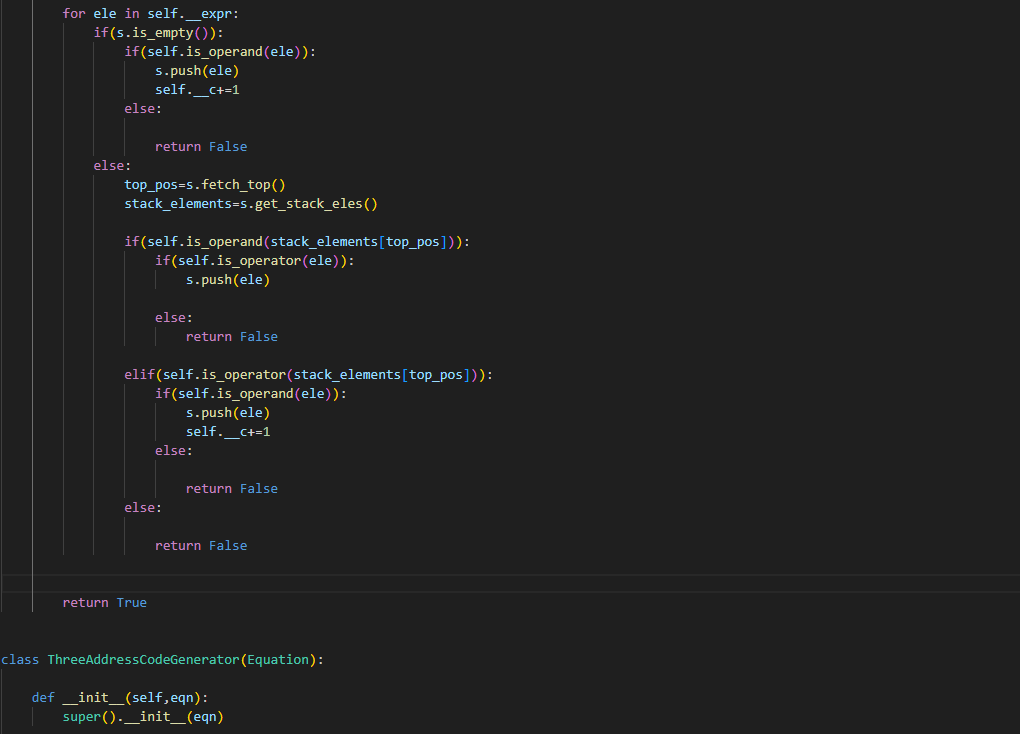
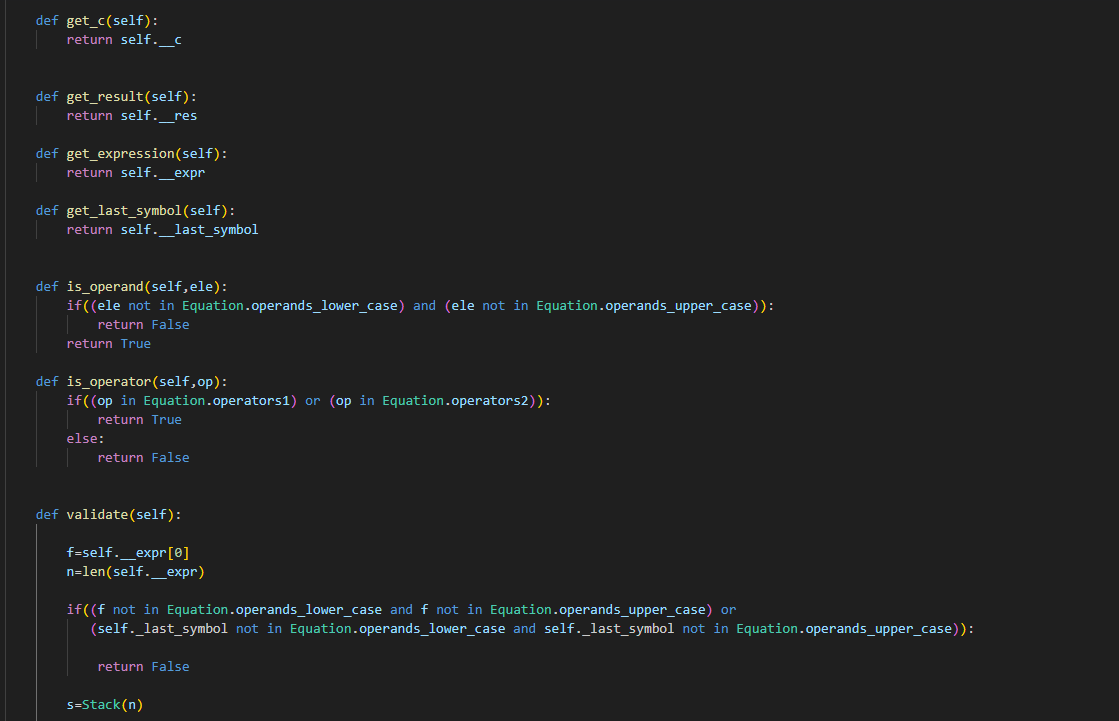
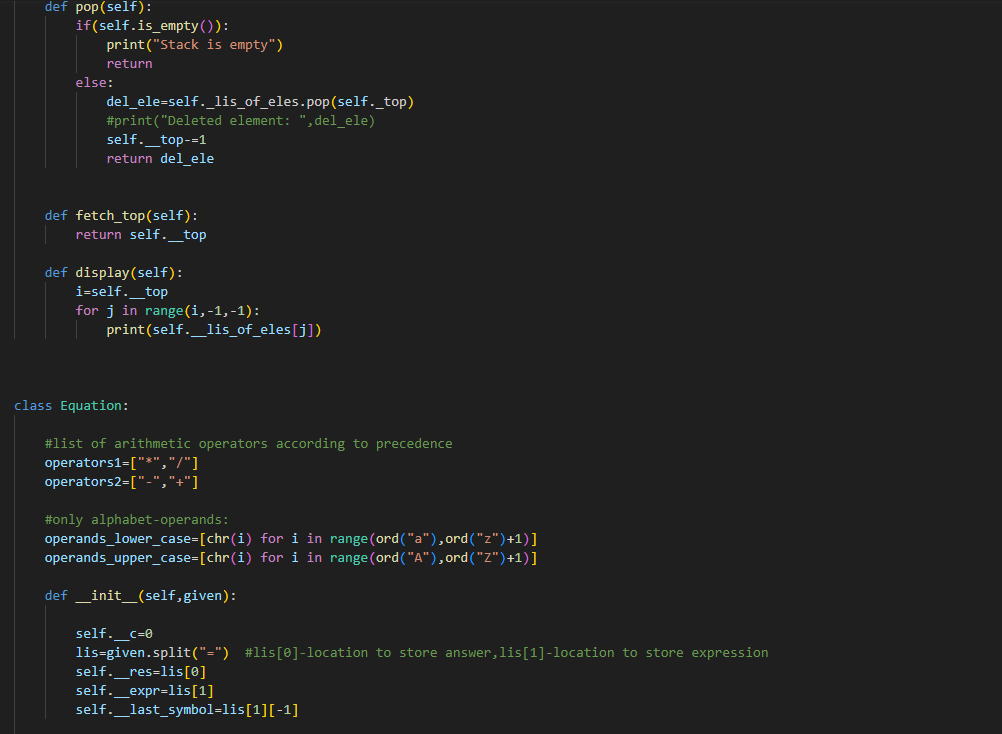
* + 1. **GUI**

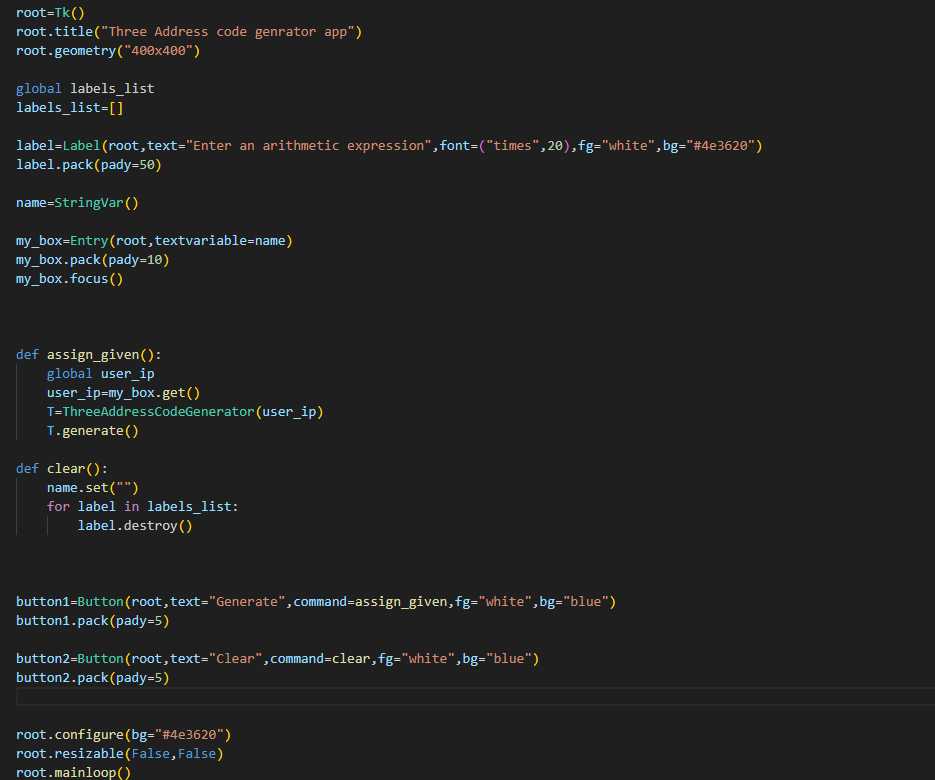
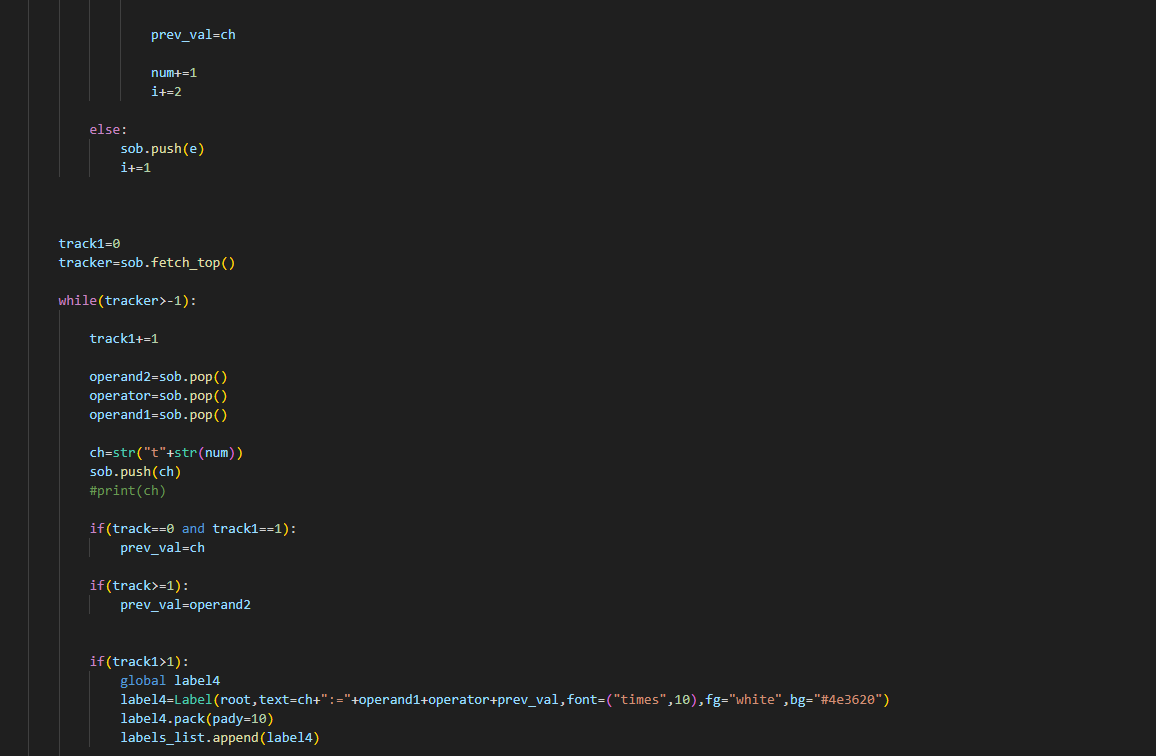
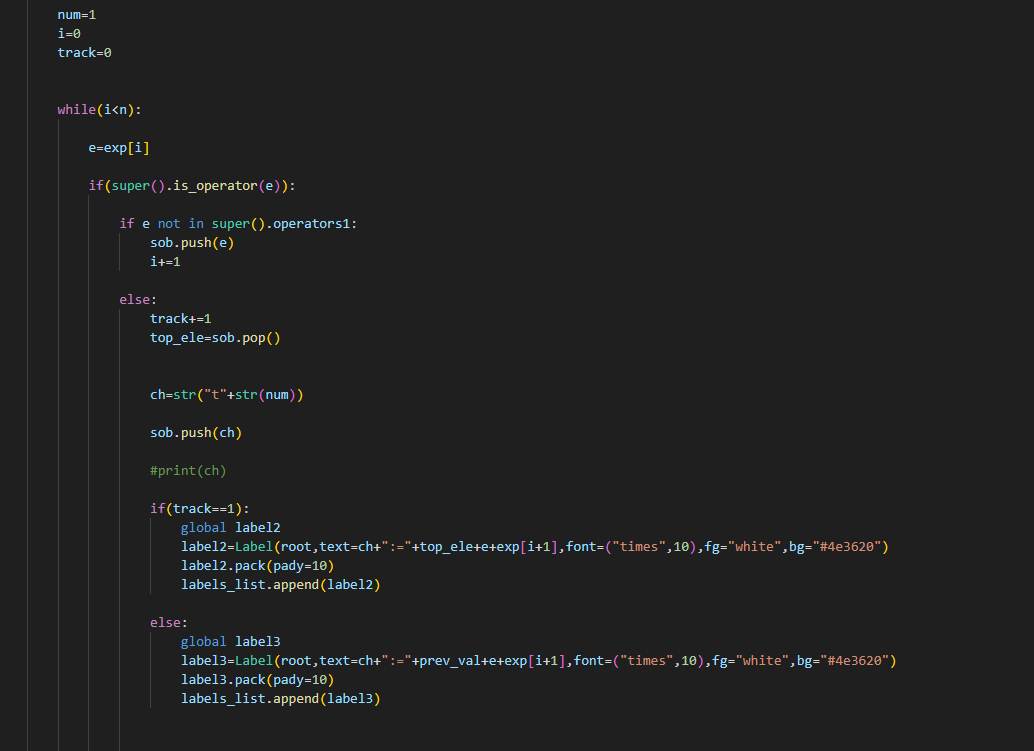
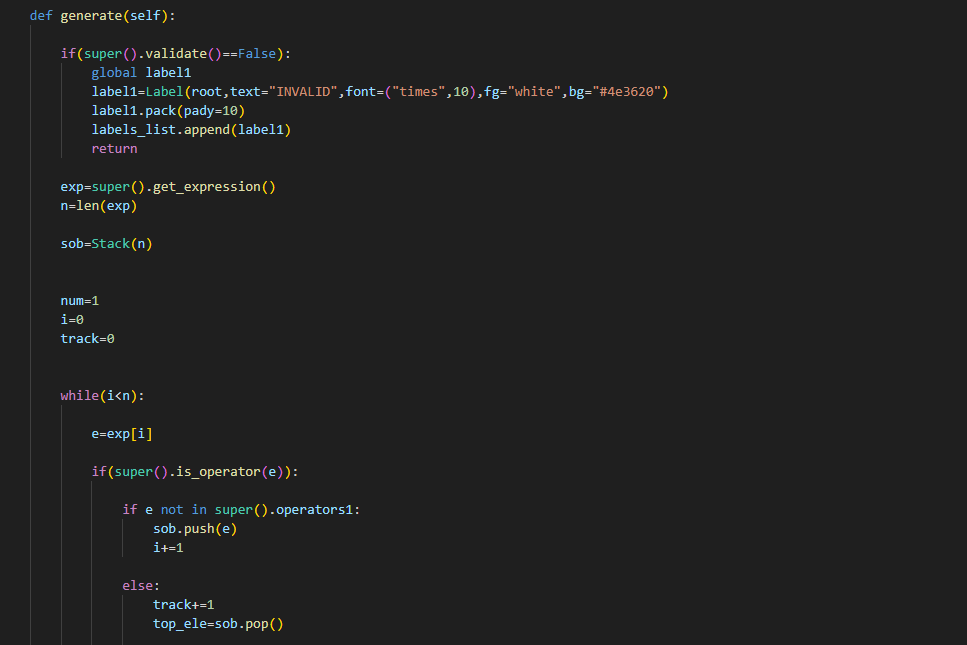
Input:



 Output:

# Code





**CHAPTER 5**

**RESULT**

The implementation of the intermediate code generation, has been successful in our project. The integration of modern web technologies, such as React JS and Node JS, has made the compiler more accessible to a wider range of users. Our compiler can effectively translate source code into executable code, making it a useful tool for developers and programmers. Through the development process, we encountered various challenges, such as ensuring the accuracy of the intermediate code generation process, but we were able to overcome these challenges through careful planning and testing. Overall, our project demonstrates our proficiency in compiler development and our ability to apply the concepts and tools learned in class to real-world applications. We believe that our compiler has the potential to be a valuable resource for the programming community and we are excited to see how it will be used in the future.

# CHAPTER 6 CONCLUSION

In conclusion, the development of a compiler that implements the various phases of the compilation process has been a challenging and rewarding experience .We now have a better knowledge of how compilers operate on a fundamental level and the significance of each step in the compilation process thanks to the implementation of the intermediate code generation. The compiler is now more accessible to a larger range of users and more user-friendly thanks to the inclusion of contemporary web technologies. We are excited to see how our compiler is utilized in the future because we think it has the potential to be a useful tool for programmers and developers.

We have learned a lot about software engineering and project management during the development process. We discovered how critical thorough planning, testing, and documentation are to a project's success. We also learned how crucial it is for teams to communicate and work together, and how productive teams may produce results that are more effective and efficient. Overall, our project has been a valuable learning experience that has allowed us to apply the concepts and tools learned in class to a real-world application. We are proud of what we have accomplished and look forward to applying our newfound knowledge to future projects and endeavors.