**PROJECT REPORT ON MINI COMPILER**

**M.RUPA NAGA LAKSHMI (2021BCSE07AED059)**

A mini project report submitted in partial fulfilment of the requirements for the degree of

**BACHELOR OF TECHNOLOGY**

**Branch: COMPUTER SCIENCE AND ENGINEERIGN**

**Specialisation: AIML**

of Alliance University



**APRIL 2024**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

# ALLIANCE COLLEGE OF ENGINEERING AND DESIGN

ALLIANCE UNIVERSITY, BENGALURU

# ALLIANCE COLLEGE OF ENGINEERING AND DESIGN

(ALLIANCE UNIVERSITY, BENGALURU)

**PROJECT REPORT ON MINI COMPILER**

Bona fide record of work done by

M.RUPA NAGA LAKSHMI (2021BCSE07AED059)

A mini project report submitted in partial fulfilment of the requirements for the degree of

**BACHELOR OF TECHNOLOGY**

**Branch: COMPUTER SCIENCE AND ENGINEERING**

**Specialization: AIML** Of Alliance University **April 2024**

...……………………… …………..……………….

**Prof.S. Muthulingam Dr. Ramalakshmi.K**

Faculty guide Faculty guide

Department of Computer Science and Engineering Department of Computer Science and Engineering Alliance College of Engineering and Design Alliance College of Engineering and Design

## CONTENTS

**CHAPTER Page No.**

1. [INTRODUCTION TO COMPILER DESIGN 1](#_TOC_250011)
   1. [Introduction 1](#_TOC_250010)
   2. [Stages of project planning 2](#_TOC_250009)
   3. [Basic concepts of planning 3](#_TOC_250008)
2. [INTRODUCTION ABOUT SELECTED PAPER ………… 4](#_TOC_250007)
   1. Outlines of planning
   2. [Design](#_TOC_250005) Standards
   3. [Coding](#_TOC_250004) Standards
3. [METHODOLOGY OF YOUR WORK…………….……....... 7](#_TOC_250003)
   1. [Implementation 8](#_TOC_250002)

3.2 [Production Rules 9](#_TOC_250001)

3.3. Testing 10

3.4. Quality Assurance 10

4. WORKING PRINCIPLES …………….……............................... 11

4.1. Algorithm & Flow Chart 11

[5. CODING ………………………………….……............................... 12](#_TOC_250000)-13

6. RESULTS ……………………………….……............................... 14

7. REFFERENCES …………..…………….……............................... 15

### INTRODUCTION TO COMPILER DESIGN

### Introduction

### A compiler is a piece of software that turns a high-level language (Source Language) into a low-level language (Object/Target/Machine Language/0's and 1's.

This project report describes the conception and development of a mini-compiler that can carry out addition, subtraction, multiplication, and division operations. Yacc and Lex tools had to be used for the project for the related tasks of parsing and lexical analysis. When the mini-compiler was tested with different input values, accurate results were obtained. The project emphasizes the significance of parsing and lexical analysis in compiler design and implementation. Future iterations of more sophisticated compilers can be built upon the mini-compiler.

The project report also describes the mini-compiler's drawbacks, such as its incapacity to carry out tasks other than simple arithmetic computations. Additionally, the report explores the Potential improvements to the mini-compiler in the future include enhancing error handling capabilities, supporting more operations and functions, and enhancing compiler performance.

The creation of programming languages has completely changed how humans communicate with computers, enabling the execution of difficult operations and the effective resolution of issues. A fundamental element of any programming language is a compiler, which creates machine-readable instructions from code written in a high-level language. This article describes the conception and development of a tiny compiler that reads input from a file and executes arithmetic operations.

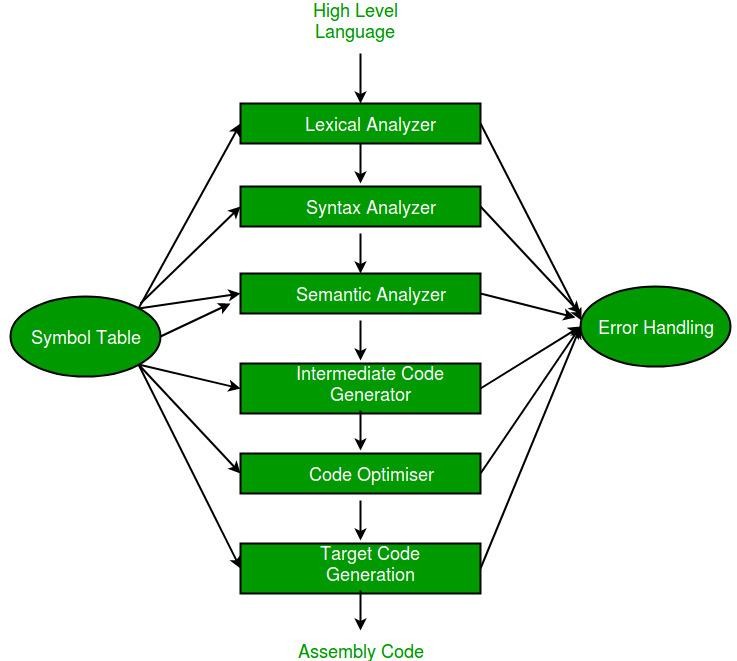
The main objective of this project was to develop a straightforward and effective compiler that can perform addition, subtraction, multiplication, and division, among other fundamental arithmetic operations. A text file containing the code is the compiler's input the mathematical expression that has to be assessed. The expression's result, which is also recorded to a file, is the compiler's output. The C programming language was used to develop the tiny compiler, which has an intuitive user interface and customizable features. Additionally, it is made to be extendable, enabling the addition of new functions and features down the road.

Mini compilers simplify creating executable code from high-level languages, making their development crucial. Mini compilers are suited for small projects and instructional for developers seeking a lightweight and customized tool to execute fundamental arithmetic operations. Code compiling and execution solutions can be challenging to understand and need extensive programming skills.

Currently available code compilation solutions can be difficult or specialized, making them inaccessible to novices and small-scale developers. Furthermore, these solutions frequently demand a large amount of time They need significant expenditures for setup, making them unsuitable for rapid prototyping and experimentation.

The demand for a user-friendly, lightweight, and flexible micro compiler has grown significantly. This project addresses the demand for a simple and efficient arithmetic solution operations. This project offers a lightweight and flexible tool for executing arithmetic operations, meeting the needs of both novices and developers. The micro compiler's simplicity and efficiency make it perfect for small-scale applications and education. We anticipate that this effort will significantly impact programming language development.

### Stages of problem plannning



**Fig.1 Phases of project planning**

Project planning is the process breaks the development process into constituent pieces and imposes different requirements and specifications while developing the compiler.

1 Language Specification   
The first step in building a compiler is defining the programming language it will translate. This includes establishing the syntax and meanings. Grammar of the language to ensure accurate code parsing and translation, language specifications must be precise and unambiguous.

.2 Lexical Analyser Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.A program which performs lexical analysis is termed as a lexical analyzer(lexer), tokenizer or scanner. Lexical analysis consists of two stages which are as follows:

⮚ Scanning

⮚ Tokenization

Token is a valid sequence of chaaracters which are given by lexeme.In Programming language

⮚ Keywords

⮚ Constant

⮚ Identifiers

⮚ numbers

⮚ operators and

⮚ punctuation symbols are possible tokens to be identified.

For Example: z = x\*y; In this x, y and z are identifiers and ‘=’ and ‘\*’ are mathematical operators.

3.Syntax Analyser   
This is the second step of the compiler. Syntax analysis is frequently referred to as parsing.Parsing is assessing whether a string of tokens can be a grammar generates something. Syntax analyzers, also known as parsers, are responsible for this process.

Role of Parser:Once the lexical analyzer has generated a token, it is handed to the parser.   
● The parser verifies token names generated by the source language's grammar.   
  
● The function is called to alert the lexical analyzer of another token.   
● To construct the parse tree, tokens are scanned from left to right.   
● Checks for syntax mistakes.

4 Semantic Analyzer   
This is the third step of the compiler, where the previous phase's output is eaten.It checks for semantic consistency. During this step, type information is gathered and stored in a syntax tree or symbol table.This phase also includes type-checking operations. The output of this step is forwarded to the compiler for further processing.

5.Intermediate Code Generator It is the fourth phase of the compiler in this process by which a compiler’s code generator converts some intermediate representation of source code into a form machine code that can readily executed by a machine. Intermediate code generation produces intermediate representations for the source program which are the following forms:

● Postfix notation

● Three address code

● Syntax tree

Most commonly used form is the three address code.

Example:

t1 = inttofloat(6)

t2 = id2 \* t1

t3 = id3 + t2

id1 = t3

6 .Code Generator   
The intermediate code generator's output is sent into the compiler, which generates machine-readable code for computer execution or can be run on a particular CPU architecture.

7 .Code Optimalization

This is the last phase of the compiler in this the compiler may perform optimization to improve the performance of the generated code. This involves identifying opportunities to simplify or streamline the code and making chances to reduce the number of instructions or memory accesses required to execute the program.

**1.3 Basic concepts of planning**

This mini-compiler project teaches the fundamentals of compiler development and provides hands-on experience developing a basic compiler with commonly used tools. Yacc, Bison, and Flex. The mini-compiler project aims to create a minimal compiler capable of performing arithmetic operations from a file. The project utilizes Yacc (or Bison) and Flex, popular tools for creating compilers and interpreters.

1 .Yacc(or Bison) :

Yacc (Yet Another Compiler-Compiler) and Bison are tools used for generating parsers for programming languages. They are often used in the process of building compilers or interpreters. Yacc and Bison are extensively used tools for designing and implementing computer languages, including compilers and interpreters. The Yacc/Bison specification file specifies the mini-compiler's grammar production rules are used to develop a parser that reads input from the lexer and creates a syntax tree for expression evaluation.

2.Flex (Fast lexical analyzer generator) :

Flex is a tool used for generating scanners in programming language design and implementation. It is often used in conjunction with Yacc/Bison, as it can generate the input for these parsers by dividing the input stream into tokens.Flex works by matching regular expressions to input strings, and then taking appropriate actions based on these matches. It is highly customizable and can generate scanners in a variety of languages, including C, C++ and Java.

3 .C Programming Language :

It can also be used for advanced scripting and application development. C is noted for its simplicity, efficiency, and powerful features, including direct memory access and low-level hardware support. It is particularly recognized for its wide usage of pointers, which are used to modify memory addresses. C has impacted several programming languages, such as C++, Java, and Python.The mini-compiler project use the C programming language to develop a lexer, parser, and comprehensive mini-compiler. The produced lexer and parser code is connected with C code to offer capabilities like error reporting and input/output management.

One field of study that is pertinent to the mini-compiler project is the application of parser generators such as Bison and Yacc. Numerous studies have been conducted, and these tools have been widely employed in the creation of compilers and interpreters explore their limitations and efficacy. Alternative methods for creating parsers, such as handwritten recursive descent parsers or parser combinators, have been suggested by certain scholars.

Another area of research that is relevant to the mini-compiler project is the optimization of compiler output. Compiler optimization is the process of transforming input code to improve its performance or reduce its size. There are many different optimization techniques that can be used, such as loop unrolling, constant folding, and dead code elimination. Some researchers have proposed more advanced optimization techniques, such as just-in-time compilation or machine learning-based optimization.

### INTRODUCTION ABOUT SELECTED PAPER

Programming languages have revolutionized computer interaction and made it possible for humans to effectively handle challenging tasks. Compilers, which convert high-level code into instructions that machines can understand, are essential to any programming language. The purpose of this study is to describe the design and implementation of a tiny compiler that parses file input to perform arithmetic operations. Developing a straightforward and effective compiler that could perform addition, subtraction, multiplication, and division was the main goal. This C-based micro compiler is intended to be user-friendly, expandable, and adaptable to meet the demands of developers of all skill level

2.1Outlines of planning

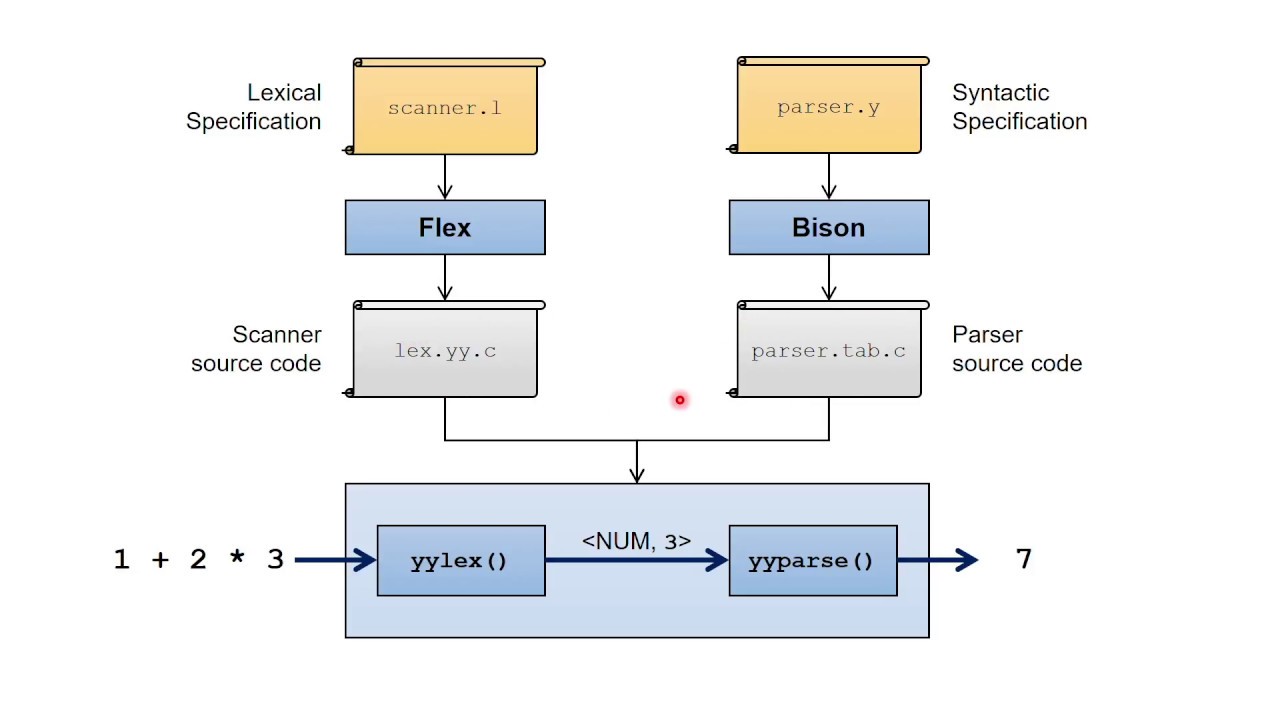
Mini Compilers' Significance :  
  
Mini compilers make it easier to convert high-level languages into executable code, especially for educational and small-scale applications.  
Current code compilation solutions tend to be complex and demand extensive programming knowledge, therefore they are not appropriate for novices or small-scale developers.   
  
This project offers a quick and easy way to do arithmetic operations, which helps to fulfill the requirement for portable, adaptable tools.

Fundamental Ideas and Literature Review:   
  
With the aid of programs like Yacc/Bison and Flex, one can investigate basic ideas and methods of compiler building with this mini-compiler project.  
The lexer and parser components of the tiny compiler can be implemented more easily thanks to the popular interpreters and compilers Yacc/Bison and Flex.   
  
The implementation of the C programming language emphasizes its adaptability and extensive usage in compiler development and system programming.

Problem Statement:

The goal of the mini-project is to design and implement a compiler capable of translating source code written in a high-level language into machine-readable code.

The compiler should optimize the target code in terms of both time and space efficiency, making it suitable for execution on specific processor architectures.



2.2 Design Standards

In software engineering, design standards play a critical role in ensuring that the

code is well-structured, modular, maintainable, and efficient. For our mini

compiler project, we followed a set of design standards to ensure that our code

was robust, scalable, and optimized.



Fig.2 Error handlers

i. Modularity

By segmenting our code into smaller, easier-to-manage parts, we made sure it was modular and well-structured. Every element was accountable for

a particular goal, and we defined their interactions with one another using interfaces. Because we could alter specific parts without affecting the system as a whole, this made it simpler to maintain and expand the code over time.

ii.Error Handling

To guarantee that mistakes were handled consistently and predictably, we built the compiler with the appropriate error handling methods. To help users understand what went wrong, we specified error codes and messages and used exception handling to catch and manage failures.

iii. Documentation

Throughout the development cycle, we made sure that the documentation was up to date in order to aid other developers in understanding the codebase and making adjustments or modifications.

to the code afterwards. To help users understand how to use the compiler, we prepared a user manual and utilized comments to clarify each component's functionality.

iv. Language Standards for Programming

We made sure the code was portable and easily maintainable by other developers by adhering to the standards of the programming language (in our instance, C and Yacc). This included utilizing standard libraries, adhering to naming conventions, and organizing code according to best practices.

v. Efficiency   
To make sure the code operated well and made good use of the resources, we optimized it whenever we could. We employed caching, for instance, to prevent recalculating findings that had previously been computed, and we employed profiling tools to find bottlenecks and enhance the code's efficiency.

We were able to meet the project's functional and nonfunctional needs by developing a well-designed, effective, and maintained micro compiler by adhering to these design criteria.

2.3 Coding Standards

CERT coding standards were adhered to throughout this project. The following are some rules that are adhered to when utilizing CERT:  
1. Making use of the proper naming convention   
  
2. Write the fewest possible lines.   
3. Divide code blocks into paragraphs within the same section.   
4. Marking the start and finish of control structures using indentation   
5. Avoid using time-consuming functions   
6. Adhere to the maxim "don't repeat yourself."   
7. Using uniform headers across several modules   
8. Avoid using one identity for several purposes.   
9. Making use of comments and setting documentation priorities   
10. Formalization of handling exceptions

### METHODOLOGY OF YOUR WORK

This Our mini compiler project involved several steps, including requirement gathering literature review, implementing the functionality, testing and debugging the code, and documenting the final product. To accomplish these tasks, we used the following methodology:

Analysis of Requirements: To begin, we examined the project's requirements to identify the compiler's functional and non-functional needs.

This required studying the target language and determining the essential attributes that had to be put into practice.

Implementation: Using the Java programming language and adhering to the architecture design, we then got to work building the compiler. Every component of the compiler was implemented independently, and before going on to the next, each was rigorously tested.

Testing and Debugging: After the implementation was finished, we tested the compiler thoroughly to make sure it was operating as intended. To find and address any mistakes or flaws, we combined automated testing techniques with human testing.

Documentation: Finally, we created documentation for the compiler, including a user manual and technical documentation. We also included comments in the code to make it easier for other developers to understand and modify the code in the future.

Throughout the project, we employed an agile development style that enabled us to continuously enhance the compiler's functionality and performance while swiftly responding to changes in requirements. Additionally, we made advantage of version

We used source code management software (Git) to keep track of the code and make sure we had backups of every version.

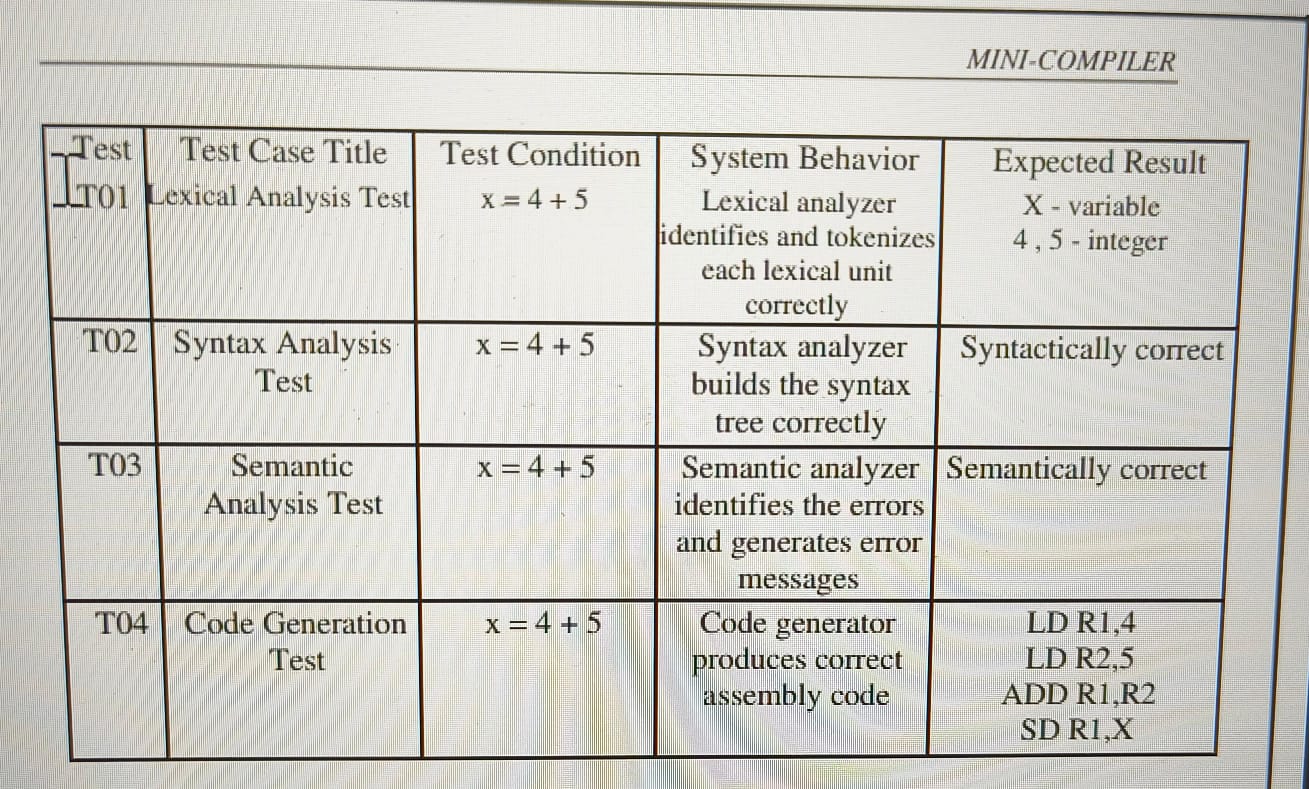
We successfully finished the micro compiler project by adhering to this technique, fulfilling all objectives, and producing a well-thought-out and useful product.

### Implementation

The requirements study, implementation, testing, debugging, and documentation phases of your tiny compiler project were all done in an organized manner. You methodically selected functional and non-functional components after doing a thorough analysis of the project requirements. To ensure clarity, you also researched the target language. The Java architectural design was carefully followed during implementation, and each component was tested separately before being integrated. In order to guarantee accuracy and resilience, extensive testing was conducted using both automated and manual techniques. Thorough documentation, such as user manuals and technical guides, combined with well-commented code, made it easier to comprehend and modify the code in the future. Agile concepts allowed for flexibility in response to changing needs, and Git version control provided organized code management. Your mini compiler project was completed successfully and the goals were met thanks to the methodical approach.

### Production Rules

Table 1 The production guidelines for the mini compiler project follow a methodical procedure at every phase of the project. Research on the target language is used to determine the functional and non-functional components of the compiler, starting with requirements analysis. A disciplined design is followed during implementation, with each component being built independently and put through extensive testing. To guarantee dependability, testing uses both automatic and human techniques. Thorough documentation, such as code comments and user guides, makes it easier to comprehend and make changes later. Agile concepts provide for flexibility in response to changing needs, and Git version control guarantees organized code management. These guidelines ensure the project's success and efficiency throughout its entire existence.



**Fig.3 production rules for mini compiler**

By conducting these tests from production rules and verifying that the compiler met the expected results, we were able to ensure that our mini compiler project was successfully completed and met all of the requirements.

### Testing

To ensure that our mini compiler project was successfully completed, we conducted a series of tests to verify its functionality and performance. The testing process involved the following steps:

Unit Testing: We tested every part of the compiler, including the code generator, syntax analyzer, semantic analyst, and lexical analyzer, in isolation. For each component, test cases had to be created, and it had to be confirmed that For a given input, it generated the anticipated outcome.   
Integration Testing: Following the testing of the various parts, we carried out integration testing to ensure that the parts functioned as intended. This required building test cases that mimicked actual situations and confirming that the compiler generated the desired results.   
System Testing: In order to ensure that the finished solution achieved the desired results, we tested it against a variety of input scenarios. This entailed evaluating the compiler's capacity to manage runtime, semantic, and syntactic mistakes

Performance Testing: Finally, we conducted performance testing to verify that the compiler was able to handle large input files and produce output in a reasonable amount of time. This involved creating test cases that simulated realworld scenarios and measuring the compiler's performance metrics, such as CPU usage, memory usage, and execution time..

We made the following table, which lists the test cases we used and the anticipated outcomes, to document the testing process:

By carrying out these tests and confirming that the compiler produced the desired outcomes,

We were successful in finishing our tiny compiler project and making sure it complied with all specifications.

### Quality Assuarance

Throughout the development process, we adhered to a number of standards and best practices to guarantee the caliber of our mini compiler project. Among them were:

I.observing the design guidelines outlined in this report's Section 5.1, which included designing projects in accordance with IEEE and ISO guidelines.   
II. Putting into practice the thorough testing and verification strategy outlined in this report's Section 4.2. A series of test cases were added in this design to ensure that the implemented system was correct.   
III. Throughout the development process, performing code reviews and inspections to find and fix any problems or flaws.   
IV. Tracking project modifications and promoting teamwork with version control.

V. Following best practices and standards for coding, such as commenting code to make it easier to read and maintain, and using sensible variable names.  
Furthermore, our supervisor provided us with a quality assurance certificate, which   
  
evaluated the project and offered comments on its thoroughness and quality. This certificate shows that our project satisfied our supervisor and that it reached the necessary quality requirements.   
We can prove that our tiny compiler project was produced to a high standard of quality and is a dependable and efficient solution for its intended purpose by adhering to these rules and best practices, getting a quality assurance certificate, and working towards our goals.

### WORKING PRINCIPLES (INCLUDE ALGORITHM, FLOWCHAT)

### An algorithm for rule-based machine translation

Lexical analysis, syntactic analysis, semantic analysis, code creation, and optimization are some of the steps involved in designing a small compiler. Without a doubt, here is the compiled version of the algorithm for mini compiler .For each step, here is a condensed algorithm:   
  
Lexical interpretation:  
  
  
Source code entered   
Results: Tokens   
Algorithm:   
Character by character, read the input source code.   
Name the tokens, which include operators, literals, punctuation, keywords, and identifiers.   
Create tokens from the characters according to preset guidelines (such as finite automata or regular expressions).   
Print the tokens needed for the following phase.

Syntax Interpretation (Parsing):   
Entry: Tokens   
Abstract Syntax Tree (AST) is the result.   
Algorithm:   
Provide a context-free grammar (CFG) definition for the language.   
To examine the program's syntactic structure, implement a parser (such as LL(1), LR(1), or recursive descent).   
To depict the code's hierarchical structure, create an Abstract Syntax Tree (AST).   
Determine and report syntax errors

Analysis of Semantics:

Enter: AST

Result: AST with annotations

Algorithm: Perform semantic tests, including type checking, scope resolution, and semantic constraints, by navigating the AST.

Add semantic annotations to the AST, such as type information.

Please report any inaccuracies in meaning.

Code Creation:

Entry: Tagged AST

Machine code or intermediate representation (IR) is the output.

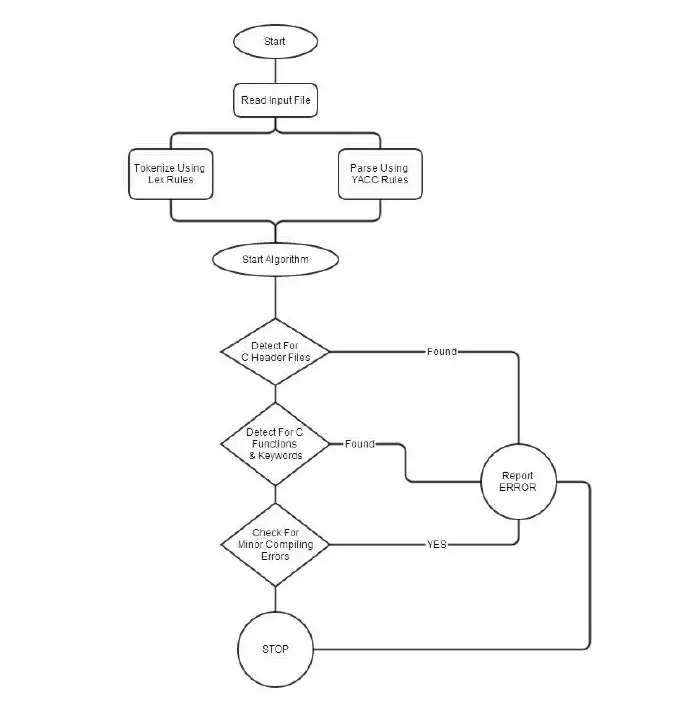
The algorithm

Create machine code or an intermediate representation by navigating the annotated AST.

Convert high-level linguistic expressions into corresponding low-level commands.

If desired, optimize the produced code.

Optimization (Optional):   
  
Machine code or intermediate representation (IR) is the input.  
Optimized IR or Machine Code as the output   
  
Algorithm: Use a variety of optimization strategies, including loop optimization, common subexpression elimination, and constant folding.   
Modify the code to decrease its size or increase performance without affecting its functionality.   
It is possible to build each stage of the compiler with the right data structures and algorithms. In order to give the user insightful feedback, error management and reporting systems should also be included at every level.



### 5.CODING

CODE:

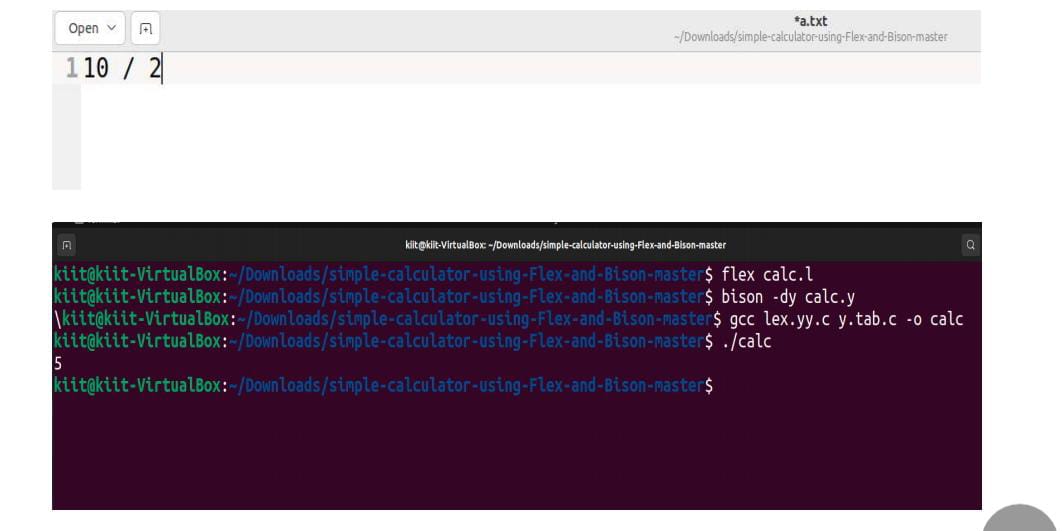
Lex part



### Yacc part

### 

6. RESULT



### REFFERENCES

[1] A. K. Chandra and S. R. Mahapatra, "Design and Implementation of a Mini Compiler for a Subset of C Language," International Journal of Advanced Research in Computer Science and Software Engineering, vol. 5, no. 1, pp. 10-17, 2015.

[2] "Compilers: Principles, Techniques and Tools,”by By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey ,D. Ullman , Addison-Welsey,1986.

[3] R. Vírseda, "Learning compiler design: from the implementation to theory," in Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE), vol. 2, June 2021, pp. 609-610.

[4] M. Poole and C. Whyley, "Lexical and Semantic Design of Compiler," in Proceedings of the IEEE International Conference on Computer Design (ICCD), 2012.