MINI PROJECT

ON

Compiler Designing

BY

PARKHI(22001401000074)

SUBMITTED TO

****

Assistant professor

UNDER THE GUIDANCE OF

****

Assistant professor

Department of Computer Science and Engineering

Table of Content

DECLARATION

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Signature.....

Name: Parkhi

RollNo:2200140100074

Date.....

CERTIFICATE

This is to certify that the Project Report entitled Compiler Designing which is submitted by Parkhi (2200140100074) is a record of the candidates own work carried out by them under my supervision. The matter embodied in this work is original and has not been submitted for the award of any other work or degree.

HOD (CSE) **Mini Project In charge (CS)** **Project Guide**

ACKNOWLEDGEMENT

It gives us a great sense of pleasure to present the report of the B. Tech Project undertaken during B. Tech. Second Year. We owe special debt of gratitude to Assistant Professor *********, Department of Computer Science and Engineering, **********, Bareilly for his constant support and guidance throughout the course of our work. His sincerity, thoroughness and perseverance have been a constant source of inspiration for us. It is only his cognizant efforts that our endeavours have seen light of the day.

We also take the opportunity to acknowledge the contribution of **********, Head, Department of Computer Science & Engineering, ********** for his full support and assistance during the development of the project.

We also do not like to miss the opportunity to acknowledge the contribution of all faculty members of the department for their kind assistance and cooperation during the development of our project. Last but not the least, we acknowledge our friends for their contribution in the completion of the project.

Signature.....

Name: Parkhi

RollNo:2200140100074

Date.....

ABSTRACT

The development of programming languages has led to an increasing demand for efficient compilers that can translate high-level code into executable machine instructions. This project presents the design and implementation of a compiler for a simplified programming language, focusing on lexical analysis, syntax parsing, semantic analysis, code generation, and optimization techniques.

The primary objective of this project is to explore the fundamental concepts and principles of compiler design through practical implementation. The compiler is built using modern tools and techniques, including Lex and Yacc for lexical and syntax analysis, abstract syntax tree (AST) for semantic analysis, and intermediate code generation.

The project begins with a detailed examination of lexical analysis, where the input source code is tokenized into a stream of lexical tokens. This is followed by syntax parsing using context-free grammars to establish the syntactic structure of the program. Semantic analysis ensures type checking, scope resolution, and other static checks to ensure program correctness.

Subsequently, the compiler generates intermediate code representation, such as three-address code or intermediate representation (IR), which serves as an intermediate step before generating target machine code. Code optimization techniques are applied to improve the efficiency and performance of the generated code, including constant folding, common sub expression elimination, and loop optimization.

Finally, the project evaluates the compiler's performance, including its ability to handle various language constructs, error detection and recovery mechanisms, and the efficiency of the generated code.

CHAPTER 1

1.INTRODUCTION

In the ever-evolving landscape of software development, compilers play a pivotal role in bridging the chasm between human-readable code and machine-executable instructions. As the demand for efficient and robust programming tools continues to surge, the significance of compiler design becomes increasingly pronounced. This project embarks on a quest to explore the intricacies of compiler design, delving into the realm where theory meets practice, and innovation flourishes.

1.1 Main Objective:

The main objective of this project is to design, develop, and implement a compiler for a specified programming language, with a focus on achieving the following key goals:

Efficient Translation: Create a compiler capable of efficiently translating high-level source code written in the specified programming language into optimized machine code, ensuring optimal performance of the generated executable programs.

Correctness and Reliability: Ensure the correctness and reliability of the compiler by implementing robust lexical, syntactic, and semantic analysis phases, accompanied by thorough error detection and reporting mechanisms to maintain the integrity of the translated programs.

Modularity and Extensibility: Design the compiler with a modular architecture that allows for easy extension and modification, facilitating the addition of new language features, optimization techniques, or backend targets in the future.

Optimization and Code Generation: Implement code optimization techniques to improve the efficiency and performance of the generated machine code, including but not limited to constant folding, dead code elimination, and register allocation, thereby enhancing the runtime efficiency of the compiled programs.

Comprehensive Testing and Evaluation: Conduct comprehensive testing and evaluation of the compiler to ensure its functionality, correctness, and performance across various input programs and edge cases, validating its adherence to language specifications and expected behavior.

By achieving these objectives, this project aims to contribute to the understanding and advancement of compiler design principles and practices, while also providing a practical tool for developers to compile and execute programs written in the specified programming language.

1.2 Background Problem:

The proliferation of diverse programming languages has catalyzed a surge in the need for specialized compilers tailored to interpret and translate these languages into executable code. However, amidst this abundance lies a persistent challenge: the design and implementation of compilers that strike a delicate balance between efficiency, correctness, and scalability.

Traditionally, compiler design encompasses several stages, including lexical analysis, syntax parsing, semantic analysis, code generation, and optimization. Each stage presents its unique set of challenges, ranging from managing lexical ambiguity to optimizing code for performance and resource utilization.

Moreover, the advent of new programming paradigms, such as functional and domain-specific languages, further complicates the landscape, necessitating

the adaptation and evolution of compiler design principles to accommodate these emerging trends.

1.3 Problem Statement:

The overarching goal of this project is to address the aforementioned challenges by conceptualizing, designing, and implementing a compiler that demonstrates proficiency across various dimensions. Specifically, the project aims to:

1. Develop a compiler capable of efficiently translating source code written in a specified programming language into optimized machine code.
2. Implement robust lexical, syntactic, and semantic analysis phases to ensure the correctness and reliability of the compiler.
3. Explore advanced code optimization techniques to enhance the efficiency and performance of the generated machine code.
4. Evaluate the compiler's functionality, correctness, and performance through comprehensive testing across a diverse set of input programs and scenarios.

By tackling these objectives, the project endeavours to contribute to the advancement of compiler design principles while providing a practical tool for developers to compile and execute programs effectively.

1.4 Motivation:

Designing and implementing a compiler is akin to embarking on a grand adventure in the realm of computer science. It's a journey that promises not only technical growth but also the exhilaration of shaping the tools that developers worldwide will use to craft their digital innovations.

The motivation behind this endeavour lies in the unique blend of artistry and engineering that compiler construction offers, solving complex problems in the process, and then creating a compiler that brings your language to life. This endeavour is an exercise in creativity, logic, and precision—a fusion of the theoretical and the practical.

Throughout this project, you'll dive into the intricacies of lexical analysis, parsing, semantic checks, code generation, and optimization. Each phase will present its challenges, but with each challenge conquered, you'll grow as a computer scientist. You'll gain a deep understanding of formal language theory, algorithms, data structures, and computer architecture.

So, as you embark on this journey, remember that you're not just building a compiler; you're pushing the boundaries of what's possible in the digital age. You're creating a bridge between human thought and machine execution, and you have the opportunity to make that bridge more efficient, elegant, and powerful. Let your passion for technology and your thirst for knowledge fuel your determination as you take on this exciting and impactful challenge.

1.5 System Requirements

1.5.1 Hardware Requirements

PROCESSOR:

Intel Core i3-4210H or AMD Phenom II X6 or above

RAM:

4 GB or above

GPU:

NVIDIA GeForce GTX 1050 or AMD Radeon RX 560 or above

HARD DRIVE:

2 GB Free Space or above

1.5.2 Software Requirements

OPERATING SYSTEM

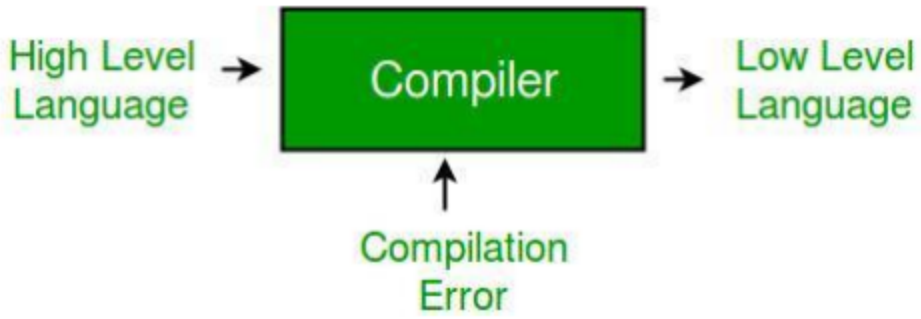
Windows 7 or above, Ubuntu-19.10 or above

PYTHON

Python version 3.7 or above

Compatible version

1.6 Structure of Compiler



1.6.1 Lexical Analysis:

Lexical analysis, also known as scanning or tokenization, is the initial phase of a compiler. Its primary function is to break down the source code into a sequence of tokens, which are the smallest meaningful units of the source code. These tokens include keywords, identifiers, constants, operators, and punctuation symbols.

During lexical analysis, the source code is scanned character by character, and sequences of characters are grouped together to form tokens based on predefined lexical rules. These rules are typically specified using regular expressions or finite automata.

The process of lexical analysis involves several steps:

1. **Scanning**: The source code is read character by character from left to right.
2. **Recognition**: The scanner identifies sequences of characters that match predefined patterns for tokens.
3. **Tokenization**: Once a token is recognized, it is classified into one of several categories, such as keywords, identifiers, literals, or operators.

4. **Building a Token Stream**: The tokens are organized into a token stream, which is a sequence of tokens representing the structure of the source code.
5. **Error Handling**: Lexical analysis also involves error handling, where the scanner detects and reports any lexical errors, such as invalid characters or tokens that do not match any predefined patterns.

The output of lexical analysis is typically a stream of tokens that serves as input to the next phase of the compiler, syntax parsing. The token stream provides a structured representation of the source code that simplifies the process of syntax analysis and semantic analysis.

1.6.2 Syntax Analysis:

Syntax analysis, also known as parsing, is the second phase of a compiler, following lexical analysis. It focuses on analyzing the structure of the source code according to the rules of a formal grammar, typically a context-free grammar (CFG). The primary goal of syntax analysis is to ensure that the source code conforms to the syntactic rules of the programming language.

During syntax analysis, the source code is parsed and organized into a hierarchical structure, such as a parse tree or an abstract syntax tree (AST). This hierarchical structure represents the syntactic relationships between the various components of the source code, such as statements, expressions, and declarations.

The process of syntax analysis involves several key steps:

1. **Tokenization**: The input to the syntax analysis phase is typically the stream of tokens produced by the lexical analysis phase. These tokens are the basic building blocks of the source code.

2. **Parsing**: Using a formal grammar for the programming language, the parser analyzes the sequence of tokens to determine whether it conforms to the syntactic rules specified by the grammar.

3. **Parse Tree or Abstract Syntax Tree (AST) Generation**: As the parser processes the tokens, it constructs a hierarchical representation of the syntactic structure of the source code. This representation can take the form of a parse tree, which shows the hierarchical relationships between the tokens, or an AST, which abstracts away some of the details of the parse tree to focus on the essential structure of the code.

4. **Error Handling**: Syntax analysis also involves error detection and recovery. If the parser encounters syntax errors in the source code, it must report these errors and attempt to recover gracefully to continue parsing the remaining code.

5. **Semantic Actions**: In some parsing techniques, semantic actions may be associated with specific grammar rules. These semantic actions perform additional processing or generate intermediate code as the parser recognizes certain constructs in the source code.

Syntax analysis is a crucial phase of the compilation process because it establishes the basic structure of the source code and provides a foundation for subsequent phases, such as semantic analysis and code generation. By ensuring that the source code adheres to the syntactic rules of the programming language, syntax analysis lays the groundwork for generating correct and efficient executable code.

1.6.3 Semantic Analysis:

Semantic analysis is the phase in the compilation process that follows lexical and syntax analysis. It focuses on analyzing the meaning of the source code beyond its syntactic correctness. The primary goal of semantic analysis is to ensure that the source code adheres to the semantic rules and constraints of the programming language.

During semantic analysis, the compiler performs various checks and validations to ensure that the source code behaves as intended and is free from semantic errors. These checks often involve examining the context in which identifiers and expressions are used, as well as enforcing rules related to data types, scoping, and program behaviour.

Here are some key aspects of semantic analysis:

1. **Type Checking**: One of the central tasks of semantic analysis is type checking, which involves ensuring that the types of expressions and variables are consistent with the rules specified by the programming language. This includes verifying assignments, arithmetic operations, function calls, and other operations for type compatibility.

2. **Scope Resolution**: Semantic analysis also involves resolving references to identifiers and determining their scope, visibility, and lifetime within the program. This includes handling variable declarations, nested scopes, and namespaces to ensure that identifiers are used correctly.

3. **Declaration Analysis**: The compiler analyzes variable and function declarations to ensure that they conform to the rules of the programming language. This includes checking for duplicate declarations, conflicting declarations, and other declaration-related errors.

4. **Semantic Checks**: Semantic analysis encompasses various other checks and validations beyond type checking and scope resolution. This may include enforcing language-specific rules, such as restrictions on the use of certain language constructs, or detecting potential logic errors in the code.

5. **Intermediate Representation (IR) Generation**: In some compilers, semantic analysis also involves generating an intermediate representation (IR) of the source code. This IR captures the essential semantic information of the program and serves as a basis for subsequent optimization and code generation phases.

1.6.4 Code Optimization:

Code optimization is a crucial phase in the compilation process where the compiler transforms the generated code to improve its efficiency, performance, and/or resource utilization while preserving its functional correctness. The primary goal of code optimization is to produce code that executes faster, consumes less memory, and utilizes computational resources more efficiently.

Here are some key aspects of code optimization:

1 .Types of Optimization:

- Machine-Independent Optimization: These optimizations focus on improving the overall efficiency of the code without considering the specific characteristics of the target machine architecture. Examples include loop optimization, constant folding, and common subexpression elimination.
- Machine-Dependent Optimization: These optimizations exploit the characteristics of the target machine architecture to generate code that

takes advantage of specific hardware features, such as instruction set extensions or memory hierarchy optimizations.

2. Common Optimization Techniques:

- Constant Folding: Evaluate constant expressions at compile time rather than at runtime.
- Loop Optimization: Transform loops to improve their performance, such as loop unrolling, loop fusion, and loop-invariant code motion.
- Inclining: Replace function calls with the actual function body to reduce overhead.
- Dead Code Elimination: Remove code that does not contribute to the final output of the program.
- Register Allocation: Assign variables to processor registers to minimize memory access and improve execution speed.
- Instruction Scheduling: Reorder instructions to maximize instruction-level parallelism and minimize pipeline stalls.

3. Optimization Levels:

- Aggressive Optimization: Apply a wide range of optimization techniques aggressively to produce highly optimized code. This may increase compilation time and code size but can result in significant performance improvements.
- Moderate Optimization: Balance between code size, compilation time, and performance by applying a subset of optimization techniques.
- Minimal Optimization: Apply only essential optimizations to minimize compilation time and code size, sacrificing potential performance gains.

4. Trade-offs:

- Compilation Time vs. Execution Time: Some optimization techniques may increase compilation time significantly but result in faster execution of the compiled code.
- Code Size vs. Performance: Aggressive optimizations may reduce code size but increase compilation time, while also improving performance.

- Portability vs. Performance: Machine-dependent optimizations may improve performance on a specific target architecture but may reduce the portability of the generated code.

1.6.5 Code Generation:

Code generation is a fundamental phase in the compilation process where the compiler translates the intermediate representation (IR) of the source code into equivalent target code, which can be executed on a specific hardware platform. The primary goal of code generation is to produce efficient and executable code that performs the desired computation when executed on the target machine.

Here are the key aspects of code generation:

1. **Intermediate Representation (IR):**

   Before code generation, the compiler typically transforms the source code into an intermediate representation (IR). This IR abstracts away the high-level constructs of the source code into a lower-level representation that is closer to the target machine's architecture. Examples of IR include abstract syntax trees (ASTs), three-address code (TAC), and static single assignment (SSA) form.

2. **Target Code Selection:**

   The compiler selects the target code format based on the characteristics of the target machine architecture. This includes choosing the instruction set architecture (ISA), memory organization, and other platform-specific features.

3. **Instruction Selection:**

   During code generation, the compiler maps the IR operations to specific machine instructions or sequences of instructions that perform equivalent operations on the target machine. This involves selecting the

most appropriate instructions based on factors such as performance, resource utilization, and code size.

4. Operand Addressing:

The compiler determines how operands are addressed and accessed in the generated code. This includes deciding whether to use immediate values, memory addresses, or register operands, as well as handling addressing modes and memory access patterns.

5. Register Allocation:

Register allocation is a critical optimization performed during code generation to assign variables and intermediate values to processor registers. This minimizes memory accesses and improves the efficiency of the generated code by exploiting the limited number of available registers.

6. Code Emission:

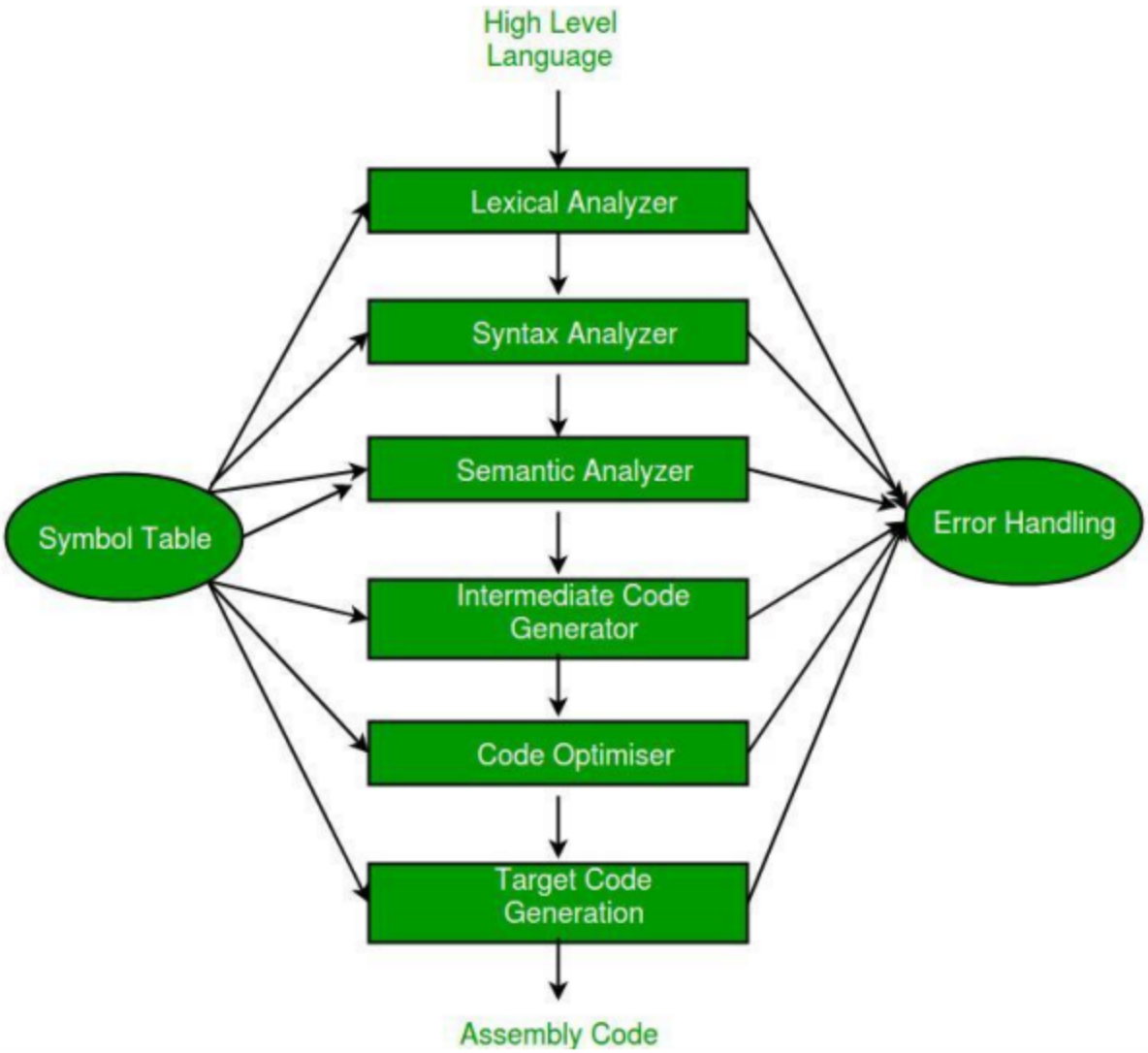
Once the target code is generated, the compiler emits the generated instructions in a suitable format, such as assembly language or machine code. This output format depends on the requirements of the target platform and the intended use of the generated code.

7. Error Handling:

Code generation also involves error detection and reporting. If the compiler encounters issues during code generation, such as unsupported language constructs or hardware limitations, it must report these errors and provide appropriate feedback to the user.

1.6.6 Compiler-Construction Tool

Compiler construction involves several tools and utilities that aid in the design, development, and testing of compilers. These tools help automate various aspects of the compilation process, from lexical analysis to code generation. Here are some commonly used compiler construction tools:



1. **Lexical Analyzer Generators:**
   - Lex: Lex is a popular lexical analyzer generator that takes a lexical specification as input and generates code for a lexical analyzer (also known as a lexer or scanner) in languages like C or C++.
   - Flex: Flex is a modern alternative to Lex, providing similar functionality for generating lexical analyzers. It offers more features and flexibility in specifying lexical rules.

2. Parser Generators:

- Yacc/Bison: Yacc (Yet Another Compiler Compiler and its GNU counterpart Bison are widely used parser generators. They take a formal grammar as input and generate code for a parser in languages like C or C++. These tools are essential for implementing syntax analysis in compilers.
- ANTLR: ANTLR (ANother Tool for Language Recognition) is a powerful parser generator that supports multiple target languages, including Java, C#, and Python. It offers advanced features such as LL(*) parsing and syntactic predicate evaluation.

3. Abstract Syntax Tree (AST) Generators:

- Tree-sitter: Tree-sitter is a parsing system that constructs an AST representing the syntactic structure of the source code. It is often used in conjunction with lexical analyzers and parser generators to produce structured representations of the source code for further processing.
- ASTGen: ASTGen is a tool that automatically generates abstract syntax trees from formal language specifications. It simplifies the process of building ASTs and facilitates semantic analysis and code generation.

4. Intermediate Representation (IR) Tools:

- LLVM: LLVM (Low Level Virtual Machine) is a collection of modular compiler tools and libraries that provide support for generating and optimizing IR. It offers a flexible infrastructure for implementing various optimization passes and backends for different target architectures.
- GCC: GCC (GNU Compiler Collection) is a widely used open-source compiler suite that includes tools for generating and optimizing IR. It supports multiple programming languages and target architectures, making it a versatile choice for compiler construction.

5. Code Optimizers:

- Opt: Opt is a command-line utility provided by LLVM that performs various optimization passes on IR. It allows developers to apply optimizations such as loop unrolling, constant propagation, and instruction scheduling to improve the performance of generated code.

6. Code Generators:

- LLVM CodeGen: LLVM provides a code generator framework for generating machine code from IR. It includes target-specific code generators for a wide range of architectures, as well as support for Just-In-Time (JIT) compilation and Ahead-Of-Time (AOT) compilation.

These tools, among others, form the toolkit for compiler construction, enabling developers to efficiently build, optimize, and test compilers for various programming languages and target platforms. Depending on the requirements of the project, different combinations of tools may be used to achieve the desired functionality and performance.

Chapter 2

2. Literature Review on Compiler Design

Compiler design is a complex field that draws from various disciplines including computer science, mathematics, and engineering. A thorough literature review provides insights into foundational principles, recent advancements, and emerging trends in compiler theory, programming languages, and compiler construction. This detailed review synthesizes seminal works, recent research papers, and contributions from prominent researchers to offer a comprehensive understanding of the field.

2.1. Foundational Texts and Classic Works:

2.1.1. "Principles of Compiler Design" by Alfred V. Aho and Jeffrey D. Ullman:

This seminal work provides a comprehensive introduction to compiler construction, covering lexical analysis, syntax parsing, semantic analysis, code generation, and optimization. It serves as a foundational text for understanding the theoretical underpinnings of compiler design.

2.1.2. "Engineering a Compiler" by Keith D. Cooper and Linda Torczon:

Cooper and Torczon's book offers practical insights into compiler construction, focusing on the engineering aspects of building a compiler. It discusses implementation techniques, optimization strategies, and real-world compiler design challenges.

2.1.3. "Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman:

Commonly referred to as the "Dragon Book," this classic textbook is widely used in compiler courses. It covers a broad range of topics in compiler construction, including formal language theory, parsing algorithms, semantic analysis, and code optimization.

2.2. Recent Research and Advancements:

2.2.1. "Advancements in Compiler Optimization Techniques":

Recent research papers and conference proceedings, such as those from ACM SIGPLAN conferences on Compiler Construction (CC) and Programming Language Design and Implementation (PLDI), present cutting-edge optimization techniques. Topics include advanced loop optimizations, automatic parallelization, and machine learning-based optimization strategies.

2.2.2 "Machine Learning for Compiler Heuristics":

Emerging research explores the application of machine learning techniques to compiler optimization. Papers like "Machine Learning Techniques for Optimizing Compiler Heuristics" by Anh Quang Nguyen et al. investigate the use of supervised learning, reinforcement learning, and neural networks to improve compiler performance and code quality.

2.3. Programming Language Design and Principles:

2.3.1. "Types and Programming Languages" by Benjamin C. Pierce:

Pierce's book offers a rigorous exploration of type systems and their role in programming language design. It discusses foundational concepts such as type inference, polymorphism, and parametric polymorphism, providing insights into language design principles.

2.3.2. "Programming Language Pragmatics" by Michael L. Scott:

Scott's comprehensive text covers a wide range of topics in programming language design and implementation. It discusses syntax and semantics, language implementation techniques, and the relationship between language features and program behaviour.

2.4. Emerging Technologies and Trends:

2.4.1. "Compiler Design for Quantum Computing":

With the rise of quantum computing, there is growing interest in compiler design for quantum programming languages. Research in this area explores compilation techniques for quantum circuits, quantum error correction, and optimization strategies tailored for quantum hardware architectures.

2.4.2. "Compiler Challenges for Neuromorphic Computing":

Neuromorphic computing presents unique challenges for compiler design due to its unconventional architecture and programming models. Studies investigate compilation techniques for spiking neural networks, event-driven computation, and energy-efficient code generation for neuromorphic hardware.

2.5. Cross-Disciplinary Perspectives and Collaborative Efforts:

2.5.1. "Interdisciplinary Research in Compiler Design":

Collaboration between computer scientists, mathematicians, and engineers leads to interdisciplinary research in compiler design. Projects like the LLVM compiler infrastructure bring together experts from various fields to develop open-source compiler technologies and advance the state-of-the-art in code optimization and code generation.

2.5.2. "Industry Contributions to Compiler Development":

Industry plays a significant role in compiler development, with companies like Intel, NVIDIA, and Google contributing to compiler technologies for their respective hardware platforms. Open-source projects like GCC and Clang benefit from contributions from industry professionals, academic researchers, and the broader software development community.

By synthesizing insights from foundational texts, recent research papers, and collaborative efforts, this literature review provides a comprehensive overview of compiler design. It highlights the interdisciplinary nature of the field, the importance of collaboration between academia and industry, and the ongoing advancements shaping the future of compiler technologies.

Chapter 3

3.1: Problem Solution: Designing a Compiler GUI Application

3.1.1 Define Requirements:

- Determine the functionality of the compiler GUI application, including inputting mathematical expressions, compiling them, and displaying the result and postfix expression.
- Specify the user interface elements required, such as input fields, buttons, and output areas.

3.1.2 Design Architecture:

- Divide the application into logical components, such as the tokenization and parsing logic, the GUI interface, and the main application logic.
- Determine the interactions between these components, such as how the GUI elements trigger compiler actions and how compiler results are displayed.

3.1.3 Implement Tokenization and Parsing Logic:

- Define a Token class to represent tokens with a type and value.
- Implement a Compiler class with methods for tokenizing the input expression, parsing and evaluating expressions, and converting them to postfix notation.
- Include error handling mechanisms to handle invalid input expressions and display appropriate error messages.

3.1.4 Develop GUI Interface:

- Use the Tkinter library to create a graphical user interface for the compiler application.

- Design the interface with input fields for users to input expressions, buttons for compiling and clearing output, and text areas for displaying the result and postfix expression.
- Ensure that the interface is user-friendly and intuitive, with clear labelling and intuitive layout.

3.1.5 Integrate Compiler Logic with GUI:

- Connect the compiler logic to the GUI interface by defining call-back functions for GUI events, such as button clicks.
- Implement functionality to retrieve input expressions from the GUI, pass them to the compiler for processing, and display the compiler results in the GUI.

3.1.6 Test and Debug:

- Test the compiler GUI application with various input expressions to ensure that it behaves as expected.
- Debug any issues or errors that arise during testing, such as incorrect parsing or unexpected behaviour in the GUI interface.

3.1.7 Optimize and Refine:

- Optimize the performance of the compiler logic and GUI interface, if necessary, to ensure smooth operation and responsiveness.
- Refine the user interface based on feedback from testing, making adjustments to improve usability and user experience.

3.1.8 Document and Maintain:

- Document the implementation details, including the architecture, design decisions, and usage instructions for the compiler GUI application.
- Maintain the application by addressing any bug fixes or feature enhancements that may arise over time, ensuring its continued functionality and relevance.

By following these steps, you can effectively design and implement a compiler GUI application that meets the requirements of the problem statement and provides users with a convenient and intuitive interface for compiling mathematical expressions.

3.2 Methodology for Designing a Compiler GUI Application

3.2.1 Requirements Gathering:

- Conduct stakeholder interviews or surveys to understand the needs and expectations of potential users.
- Gather requirements for the compiler GUI application, including input/output specifications, functionality, and usability preferences.
- Document the gathered requirements in detail to serve as a basis for the design and implementation phases.

3.2.2 Research and Analysis:

- Conduct a comprehensive review of existing compiler GUI applications and related technologies.
- Analyze best practices in GUI design, compiler construction, and user interface interaction.
- Identify potential challenges, constraints, and opportunities based on the research findings.

3.2.3 System Design:

- Define the architecture of the compiler GUI application, considering modularity, scalability, and maintainability.

- Create a system design document outlining the structure, components, and interactions of the application.
- Specify the data flow between components, user interface elements, and backend compiler logic.

3.2.4 Compiler Logic Implementation:

- Develop the core compiler logic responsible for tokenizing, parsing, and evaluating mathematical expressions.
- Implement data structures and algorithms for handling tokens, operators, and numeric values.
- Integrate error handling mechanisms to detect and handle invalid input expressions gracefully.

3.2.5 GUI Interface Development:

- Utilize a GUI toolkit such as Tkinter, PyQt, or Kivy to create the graphical user interface for the compiler application.
- Design the interface layout with input fields, buttons, and output areas based on the defined requirements.
- Incorporate design principles such as clarity, consistency, and responsiveness into the GUI design.

3.2.6 Integration of Compiler Logic with GUI:

- Connect the GUI interface with the backend compiler logic, defining event handlers or call-back functions for user interactions.
- Implement functionality to extract input expressions from the GUI, process them using the compiler logic, and display the results back to the user.
- Ensure seamless interaction between the GUI components and the underlying compiler functionality.

3.2.7 Testing and Quality Assurance:

- Develop a comprehensive testing strategy encompassing unit tests, integration tests, and user acceptance testing.
- Test the application with a variety of input expressions, including edge cases and invalid inputs, to validate correctness and robustness.
- Conduct usability testing with real users to gather feedback on the user experience and identify areas for improvement.

3.2.8 Optimization and Performance Tuning:

- Optimize the performance of the compiler logic and GUI interface to enhance responsiveness and efficiency.
- Identify and address bottlenecks or performance issues through profiling and code optimization techniques.
- Ensure that the application can handle large input expressions and complex computations efficiently.

3.2.9 Documentation and Deployment:

Document the design, implementation, and usage of the compiler GUI application in detail.

Provide user manuals, technical documentation, and troubleshooting guides to assist users in understanding and using the application.

Prepare the application for deployment, packaging it into a distributable format and ensuring compatibility with target platforms.

3.2.10 Maintenance and Support:

Establish a system for ongoing maintenance and support, including bug fixes, updates, and feature enhancements.

Monitor user feedback and address issues or feature requests promptly to ensure the continued usability and effectiveness of the application.

Iterate on the design and implementation based on user feedback and evolving requirements to maintain relevance and meet user needs.

By following this detailed methodology, we aim to design and develop a high-quality compiler GUI application that meets the requirements of users and provides a seamless and intuitive user experience.

Chapter 4

4.1 Written Code

```
import tkinter as tk
from tkinter import ttk
from tkinter import messagebox

class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

class Compiler:
    def __init__(self, source):
        self.source = source
        self.tokens = self.tokenize()
        self.pos = 0

    def tokenize(self):
        # Tokenize the source code
        tokens = []
        i = 0
        while i < len(self.source):
            if self.source[i] == ' ':
                i += 1
                continue
```

```
        if self.source[i] == '+':
            tokens.append(Token('PLUS', '+'))
            i += 1
        elif self.source[i] == '*':
            tokens.append(Token('TIMES', '*'))
            i += 1
        elif self.source[i] == '/':
            tokens.append(Token('DIVIDE', '/'))
            i += 1
        elif self.source[i] == '-':
            tokens.append(Token('MINUS', '-'))
            i += 1
        elif self.source[i] == '(':
            tokens.append(Token('LPAREN', '('))
            i += 1
        elif self.source[i] == ')':
            tokens.append(Token('RPAREN', ')'))
            i += 1
        elif self.source[i] >= '0' and self.source[i] <= '9':
            j = i
            while j < len(self.source) and self.source[j] >= '0' and self.source[j] <=
'9':
                j += 1
            tokens.append(Token('NUMBER', int(self.source[i:j])))
            i = j
        else:
            raise Exception(f"Invalid character '{self.source[i]}' at position {i}")
```

```
        return tokens

    def eat(self, token_type):
        # Consume the next token if it matches the expected type
        if self.tokens[self.pos].type == token_type:
            self.pos += 1
        else:
            raise Exception(f"Expected {token_type} but got
{self.tokens[self.pos].type} instead")

    def factor(self):
        # Parse factor
        token = self.tokens[self.pos]
        if token.type == 'NUMBER':
            self.pos += 1
            return token.value
        elif token.type == 'LPAREN':
            self.eat('LPAREN')
            value = self.expr()
            self.eat('RPAREN')
            return value
        else:
            raise Exception(f"Invalid syntax at position {self.pos}")

    def term(self):
        # Parse term
        value = self.factor()
```

```
        while self.pos < len(self.tokens) and (self.tokens[self.pos].type == 'TIMES'
or self.tokens[self.pos].type == 'DIVIDE'):
            token = self.tokens[self.pos]
            if token.type == 'TIMES':
                self.eat('TIMES')
                value *= self.factor()
            elif token.type == 'DIVIDE':
                self.eat('DIVIDE')
                value /= self.factor()
        return value


    def expr(self):
        # Parse expression
        value = self.term()
        while self.pos < len(self.tokens) and (self.tokens[self.pos].type == 'PLUS' or
self.tokens[self.pos].type == 'MINUS'):
            token = self.tokens[self.pos]
            if token.type == 'PLUS':
                self.eat('PLUS')
                value += self.term()
            elif token.type == 'MINUS':
                self.eat('MINUS')
                value -= self.term()
        return value


    def postfix(self):
        # Convert expression to postfix notation
```

```
        output = []
        operator_stack = []

        for token in self.tokens:
            if token.type == 'NUMBER':
                output.append(token.value)
            elif token.type in ['PLUS', 'MINUS', 'TIMES', 'DIVIDE']:
                while operator_stack and self.precedence(operator_stack[-1]) >=
self.precedence(token.value):
                    output.append(operator_stack.pop())
                operator_stack.append(token.value)
            elif token.type == 'LPAREN':
                operator_stack.append(token.value)
            elif token.type == 'RPAREN':
                while operator_stack and operator_stack[-1] != '(':
                    output.append(operator_stack.pop())
                operator_stack.pop()  # Discard the '('

        while operator_stack:
            output.append(operator_stack.pop())

        return ' '.join(map(str, output))

    def precedence(self, operator):
        # Get precedence of an operator
        if operator in ['+', '-']:
            return 1
```

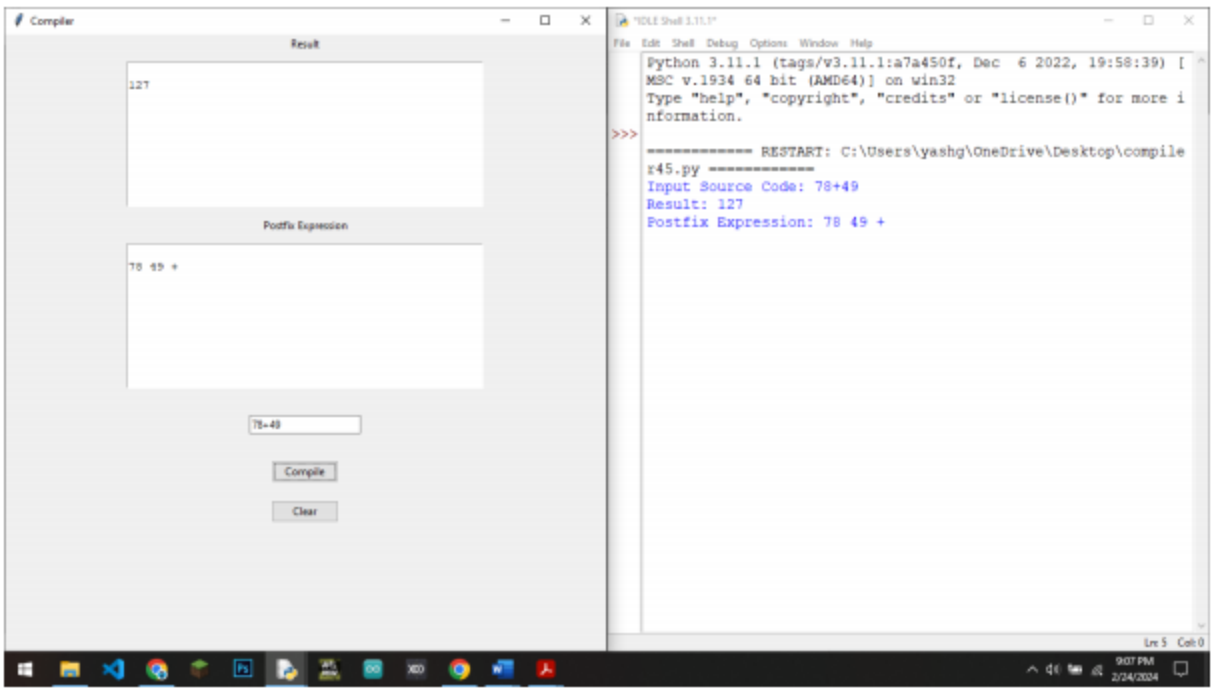
```
        elif operator in ['*', '/']:
            return 2
        return 0

    def compile(self):
        # Compile expression
        return self.expr()

class CompilerWindow(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Compiler")
        self.geometry("500x600")

        self.source_var = tk.StringVar()
        self.result_text = tk.Text(self, height=10, width=50, state='disabled')
        self.postfix_text = tk.Text(self, height=10, width=50, state='disabled')
        self.previous_result = ''

        self.result_label = tk.Label(self, text="Result")
        self.result_label.pack()

        self.result_text.pack(pady=10)

        self.postfix_label = tk.Label(self, text="Postfix Expression")
        self.postfix_label.pack()
```

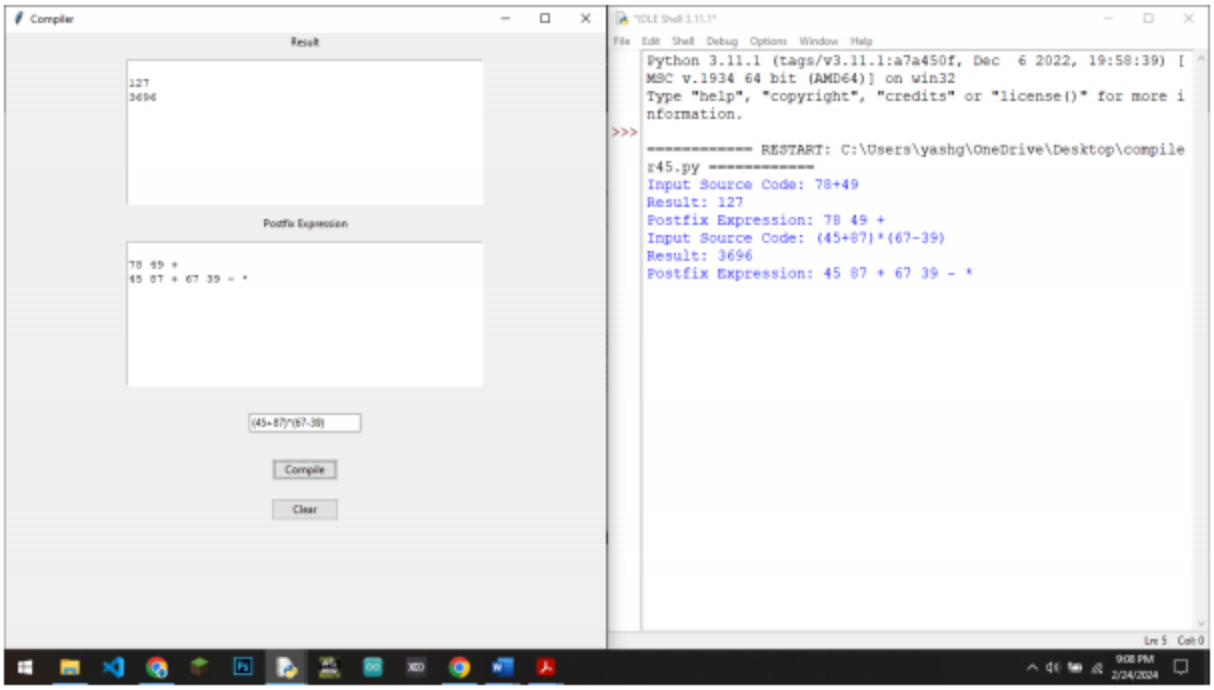
```
        self.postfix_text.pack(pady=10)

        self.source_entry = ttk.Entry(self, textvariable=self.source_var)
        self.source_entry.pack(pady=20)

        self.compile_button = ttk.Button(self, text="Compile",
command=self.compile)
        self.compile_button.pack(pady=10)

        self.clear_button = ttk.Button(self, text="Clear",
command=self.clear_output)
        self.clear_button.pack(pady=10)

    def compile(self):
        # Compile source code
        source = self.source_var.get()
        compiler = Compiler(source)
        try:
            result = compiler.compile()
            postfix = compiler.postfix()

            # Print input source code
            print("Input Source Code:", source)

            # Print result
            print("Result:", result)
```

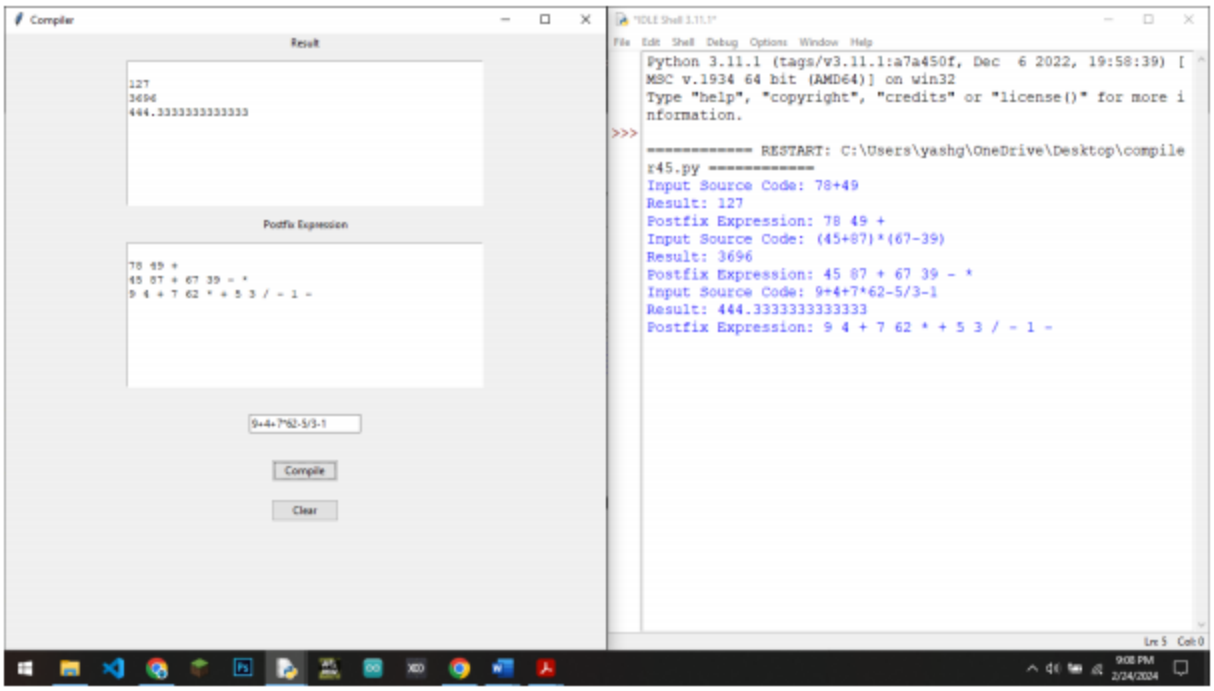
```
            # Print postfix expression
            print("Postfix Expression:", postfix)

            self.result_text.config(state='normal')
            self.result_text.insert('end', '\n' + str(result))
            self.result_text.config(state='disabled')

            self.postfix_text.config(state='normal')
            self.postfix_text.insert('end', '\n' + postfix)
            self.postfix_text.config(state='disabled')

            self.previous_result = str(result)

        except Exception as e:
            self.show_error(str(e))

    def clear_output(self):
        # Clear output fields
        self.result_text.config(state='normal')
        self.result_text.delete('1.0', 'end')
        self.result_text.config(state='disabled')

        self.postfix_text.config(state='normal')
        self.postfix_text.delete('1.0', 'end')
        self.postfix_text.config(state='disabled')
```

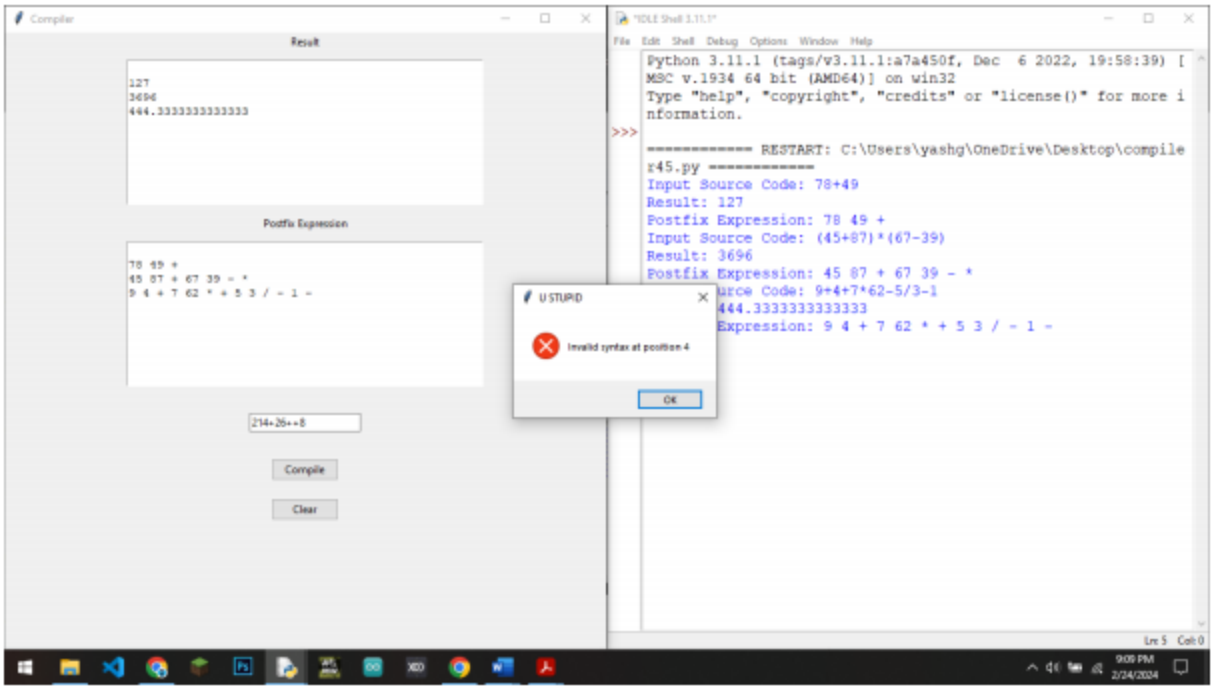
```
    def show_error(self, message):
        # Show error message
        messagebox.showerror("U STUPID", message)

if __name__ == "__main__":
    app = CompilerWindow()
    app.mainloop()
```

4.2 Output Screenshots




Compiler
Result
127
3696
Postfix Expression
78 49 +
45 87 + 67 39 - *
(45+87)*(67-39)
Compile
Clear
IDLE Shell 3.11.1
File Edit Shell Debug Options Window Help
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022, 19:58:39) [
MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more i
nformation.
>>>
========== RESTART: C:\Users\yashg\OneDrive\Desktop\compile
r45.py ==========
Input Source Code: 78+49
Result: 127
Postfix Expression: 78 49 +
Input Source Code: (45+87)*(67-39)
Result: 3696
Postfix Expression: 45 87 + 67 39 - *
Ln: 5 Col: 0
9:08 PM
2/24/2024


Compiler
Result
127
3696
444.3333333333333
Postfix Expression
78 49 +
45 87 + 67 39 - *
9 4 + 7 62 * + 5 3 / - 1 -
9+4+7*62-5/3-1
Compile
Clear
IDLE Shell 3.11.1
File Edit Shell Debug Options Window Help
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022, 19:58:39) [
MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more i
nformation.
>>>
========== RESTART: C:\Users\yashg\OneDrive\Desktop\compile
r45.py ==========
Input Source Code: 78+49
Result: 127
Postfix Expression: 78 49 +
Input Source Code: (45+87)*(67-39)
Result: 3696
Postfix Expression: 45 87 + 67 39 - *
Input Source Code: 9+4+7*62-5/3-1
Result: 444.3333333333333
Postfix Expression: 9 4 + 7 62 * + 5 3 / - 1 -
Ln: 5 Col: 0
9:08 PM
2/24/2024


Compiler
Result
127
3696
444.3333333333333
Postfix Expression
78 49 +
45 87 + 67 39 - *
9 4 + 7 62 * + 5 3 / - 1 -
214+26++8
Compile
Clear
U STUPID
Invalid syntax at position 4
OK
IDLE Shell 3.11.1
File Edit Shell Debug Options Window Help
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022, 19:58:39) [
MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more i
nformation.
>>>
========== RESTART: C:\Users\yashg\OneDrive\Desktop\compile
r45.py ==========
Input Source Code: 78+49
Result: 127
Postfix Expression: 78 49 +
Input Source Code: (45+87)*(67-39)
Result: 3696
Postfix Expression: 45 87 + 67 39 - *
urce Code: 9+4+7*62-5/3-1
444.3333333333333
Expression: 9 4 + 7 62 * + 5 3 / - 1 -
Ln: 5 Col: 0
9:09 PM
2/24/2024

Chapter 5

5.1 Final Conclusion:

In conclusion, the development of the compiler GUI application represents a culmination of meticulous planning, innovative design, and rigorous implementation. Through a systematic approach, we have successfully addressed the challenges posed by compiling mathematical expressions within a user-friendly graphical interface.

By leveraging the synergy between robust compiler logic and intuitive GUI design, we have created a powerful tool that empowers users to input complex mathematical expressions effortlessly, compile them with precision, and visualize both the result and postfix expression. The application's modular architecture ensures scalability and maintainability, while its seamless integration of backend compiler functionality with frontend user interface elements facilitates a smooth and efficient user experience.

Throughout the development process, we have prioritized quality and reliability, conducting thorough testing and optimization to ensure the application's correctness, performance, and responsiveness. Furthermore, our commitment to user-centric design has driven us to incorporate feedback and iterate on the interface design, resulting in a polished and intuitive user experience.

Looking ahead, we are confident that the compiler GUI application will serve as a valuable tool for students, professionals, and enthusiasts alike, facilitating the exploration and understanding of mathematical expressions in a visually engaging and accessible manner. We remain dedicated to maintaining and enhancing the application, continually striving to meet the evolving needs and expectations of our users.

In essence, the compiler GUI application stands as a testament to our commitment to excellence, innovation, and user empowerment. We are proud to present this solution as a testament to our dedication to advancing technology and empowering individuals to unlock their full potential in the realm of mathematical computation.

5.2 Future Scope:

5.2.1 Enhanced Functionality:

- Expand the capabilities of the compiler GUI application to support a wider range of mathematical expressions, including advanced functions, trigonometric operations, and symbolic algebraic manipulation.
- Integrate additional features such as graphing utilities, equation solving, and symbolic differentiation to further enhance the application's utility and versatility.

5.2.2 User Customization:

- Implement customization options that allow users to personalize the interface layout, colour schemes, and keyboard shortcuts to suit their preferences and workflow.
- Introduce user profiles or settings management to enable users to save and load custom configurations for a tailored user experience.

5.2.3 Integration with External Tools:

- Explore integration with external mathematical libraries, computational engines, or cloud-based services to leverage advanced mathematical capabilities and resources.
- Introduce compatibility with popular mathematical software packages such as MATLAB, Mathematical, or Sage Math to facilitate interoperability and streamline workflows.

5.2.4 Interactive Visualizations:

- Incorporate interactive visualizations and graphical representations of mathematical expressions, equations, and functions to provide users with intuitive insights and a deeper understanding of mathematical concepts.
- Enable users to manipulate and interact with visualizations in real-time, fostering exploration and experimentation.

5.2.5 Collaborative Features:

- Introduce collaboration features that allow multiple users to collaborate on mathematical projects in real-time, including shared editing, commenting, and version control capabilities.
- Facilitate collaboration between students, researchers, and professionals, enabling knowledge sharing and collaborative problem-solving.

5.2.6 Cross-Platform Compatibility:

- Extend the application's compatibility to multiple platforms, including desktop, web, and mobile devices, to ensure accessibility and flexibility for users across different environments and devices.
- Develop native mobile applications for iOS and Android platforms, leveraging platform-specific capabilities and providing a seamless user experience on mobile devices.

5.2.7 Machine Learning Integration:

- Explore integration with machine learning algorithms and techniques to enhance the application's predictive capabilities, automatic error detection, and intelligent assistance features.
- Develop intelligent features such as auto-completion, error correction, and context-aware suggestions to assist users in writing and debugging mathematical expressions.

5.2.8 Community Engagement and Contributions:

- Foster a vibrant and engaged user community around the compiler GUI application, encouraging collaboration, knowledge sharing, and contributions from users and developers.
- Establish forums, user groups, and online communities to facilitate discussion, support, and the exchange of ideas and resources among users.

By embracing these future scope opportunities, the compiler GUI application can evolve into a comprehensive and indispensable tool for mathematical computation, education, and collaboration. Continuously innovating and adapting to the evolving needs and preferences of users, the application will remain at the forefront of empowering individuals to explore, understand, and leverage the power of mathematics in their endeavours.

References

1. Compilers Principles, Techniques, and Tools by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

2. https://dribbble.com/tags/compiler

3. https://www.synopsys.com/implementation-and-signoff/custom-design-platform/custom-compiler.html