Kotha Compiler

Submitted By

Student Name	Student ID
Aupurba Sarker	232-15-269
Labony Sur	232-15-473
Alzubair Ahamed	232-15-591

MINI LAB PROJECT REPORT

This Report Presented in Partial Fulfillment of the course

CSE-314: Compiler Design lab in the Computer Science and Engineering Department

DAFFODIL INTERNATIONAL UNIVERSITY

Dhaka, Bangladesh

10 December 2025

DECLARATION

We hereby declare that this lab project has been done by us under the supervision of **Mr. Z N M Zarif Mahud**, **lecturer,** Department of Computer Science and Engineering, Daffodil International University. We also declare that neither this project nor any part of this project has been submitted elsewhere as lab projects.

Submitted To:

_____

Mr. Z N M Zarif Mahud(ZZM)
Lecturer
Department of Computer Science and Engineering
Daffodil International University

Submitted by:

Name	ID	Department	Institution	Signature
Aupurba Sarker	232-15-269	CSE	DIU	
Labony Sur	232-15-473	CSE	DIU	
Alzubair Ahamed	232-15-592	CSE	DIU	

COURSE & PROGRAM OUTCOME

The following course have course outcomes as following:

Table 1: Course Outcome Statements

CO's	Statements
CO1	Understand the working procedure of a compiler for debugging programs.
CO2	Analyze the role of syntax and semantics of Programming languages in compiler construction.
CO3	Apply the techniques, algorithms, and different tools used in Compiler Construction in the design and construction of the phases of a compiler's components.
CO4	Analyze and apply code optimization techniques to improve efficiency and performance of compiled code.

Table 2: Mapping of CO, PO, Blooms, KP, CEP and CEA

CO	PO	Blooms	KP	CEP	CEA
CO1	PO1	C1, C2	KP3	EP1	EA1
CO2	PO3	C2	KP4	EP3	EA2
CO3	PO5	C4, A1	KP6	EP5	EA3
CO4	PO4	C3, C6, A3,P3	KP2	EP3	EA5

Table of Contents

Chapter 1

Introduction

Language is the primary vessel of human thought, yet the world of computer science is overwhelmingly dominated by English. This creates a significant cognitive barrier for non-native speakers, particularly in Bangladesh, where students often struggle to map their native logic to English syntax.
Kotha is a statically-typed, high-performance programming language designed to bridge this gap. It utilises a "Banglish" syntax—Bengali keywords written in English script (e.g., dhoro for variable declaration, jodi for conditionals). Unlike simple educational toys, Kotha is a robust system featuring a full compilation pipeline, a custom Virtual Machine (VM) with Garbage Collection, and a standard library written in C.
This project goes beyond language design; it includes a complete development ecosystem featuring a web-based Integrated Development Environment (IDE) with a retro "Windows 95" aesthetic, syntax highlighting, and real-time error reporting.
by English. This creates a significant cognitive barrier for non-native speakers, particularly in Bangladesh, where language is the primary vessel of human thought, yet the world of computer science is overwhelmingly dominated.

1.1 Motivation

The motivation for developing **Kotha** stems from a critical observation in computer science education: language barriers often become cognitive barriers. For native Bengali speakers, learning to code involves two simultaneous challenges—mastering programming logic and navigating the English syntax of standard languages like C or Python.

Key motivators for this project include:

1. **Cultural & Educational Accessibility:** To democratize access to computer science by allowing students to write code using familiar concepts (e.g., jodi instead of if, dhoro instead of let). This reduces the initial friction of learning syntax.
2. **Demystifying Compilers:** In many academic curriculums, compilers are treated as "black boxes." By building a full-stack compiler from scratch—including a custom Virtual Machine and Garbage Collector—this project serves as a transparent educational tool for understanding low-level system design.
3. **Technical Exploration:** To prove that a localised language can be just as robust as standard ones. Unlike simple transpilers, Kotha implements a complete compilation pipeline (Lexer $\rightarrow$ Parser $\rightarrow$ IR $\rightarrow$ Optimization

$\rightarrow$ VM), demonstrating that "Banglish" code can be compiled into high-performance bytecode.

1.2 Objectives

The primary objective is to design and implement a statically typed, interpreted programming language with Bengali syntax. The specific technical objectives are:

1. **Develop a Full Compiler Pipeline:** To implement the six standard phases of compilation (Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Optimization, and Code Generation) using C, Flex, and Bison.
2. **Create a Custom Virtual Machine (VM):** To build a stack-based VM capable of executing custom bytecode, managing function call frames, and performing automatic memory management via Garbage Collection.
3. **Design an Intermediate Representation (IR):** To implement a 3-Address Code (3AC) system that allows for code optimization (like constant folding and strength reduction) before execution.
4. **Build a Developer Ecosystem:** To create a user-friendly, web-based Integrated Development Environment (IDE) that features syntax highlighting, file management, and real-time error reporting to make the language immediately usable.
5. **Standard Library Implementation:** To provide essential built-in functions for File I/O, String manipulation, and Mathematics, making the language capable of solving real computational problems.

1.3 Gap Analysis

This analysis compares the current landscape of programming tools with the solutions offered by Kotha.

Factor	Existing State (Traditional Languages)	Desired State (Kotha)
Syntax	English-centric (e.g., int, while). Creates a barrier for non-native speakers.	Banglish (e.g., dhoro, jotokkhon). Aligns with the natural thought process of Bengali students.
Architecture	Often closed-source or too complex for students to dissect (e.g., JVM, V8 Engine).	Transparent & Educational. A custom VM and Compiler built from scratch to be studied and modified.
Execution	Typically compiles to machine code (hard to debug) or interprets slowly.	Optimized Bytecode. Uses an Intermediate Representation (IR) to balance performance and debuggability.
Tooling	Requires complex setup (installing MinGW, setting paths) which discourages beginners.	Instant Web IDE. A "batteries-included" web interface allows coding immediately without installation.

1.4 Project Outcome

The Kotha project has successfully delivered a functional programming language ecosystem (Version 0.2.0). The specific outcomes achieved include:

1. **Functional Compiler:** A robust CLI tool (./kotha) that successfully compiles source code into bytecode or transpiles it to C.
2. **Banglish Syntax Support:** Complete support for variables (dhoro), conditionals (jodi/othoba), loops (cholbe), and functions (kaj).
3. **Virtual Machine with GC:** A working stack-based VM that executes code and manages memory automatically using a Mark-and-Sweep Garbage Collector.
4. **Web IDE:** A deployed "Windows 95" themed web interface that allows users to write, run, and save Kotha programs in the browser.
5. **Standard Library:** A set of functional native libraries for Math, String, and File operations, enabling users to write complex algorithms like Factorials or Word Frequency counters.

1.5 Current State of Kotha

At present, **Kotha (Version 0.2.0)** stands as a fully functional, statically-typed interpreted programming language. It has evolved beyond a theoretical concept into a working software ecosystem. The current system features a robust "Banglish" syntax that successfully abstracts complex C-like programming constructs into Bengali.

Technically, the project functions as a dual-mode system: it can either compile source code into bytecode for its custom Virtual Machine or transpile it directly into C for native execution. The ecosystem includes a live web-based IDE that allows users to write, debug, and run code instantly without local installation. While fully capable of handling algorithms (like Factorial or String Reversal) and basic I/O operations, it currently focuses on imperative programming paradigms and does not yet support Object-Oriented Programming (OOP) classes or structs.

1.6 Desired Future State

The vision for Kotha extends into becoming a general-purpose language suitable for both education and simple application development. The desired future state includes:

- **Object-Oriented Capabilities:** Introducing class (shreni) and object (bostu) constructs to teach modern software design patterns.
- **Advanced Developer Tooling:** Integrating a Language Server Protocol (LSP) to enable autocomplete and syntax checking in professional editors like VS Code.
- **Cross-Platform Mobile App:** Porting the IDE logic to Android and iOS, allowing students in rural areas to practice coding directly on smartphones.
- **Foreign Function Interface (FFI):** A mechanism to call external C libraries, enabling Kotha to create graphics, handle networking, or interact with hardware.

1.7 Identified Gaps

Gap analysis highlights the difference between Kotha's current capabilities and standard production languages:

Type System	Supports int, float, string.	Lacks boolean keywords (true/false) and user-defined types (structs).
Error Handling	Detects syntax errors and some runtime errors (e.g., division by zero).	Error messages are generic. It lacks a detailed stack trace to pinpoint exactly where logical errors occur.
Modularity	Supports functions (kaj) within a single file.	Lacks an import system to use code across multiple files or modules.
Optimization	Basic constant folding and strength reduction.	Lacks advanced optimization like dead-code elimination or loop unrolling.

1.8 Functional Achievements

The project has successfully delivered the following functional milestones:

1. **Banglish Syntax Implementation:** Successfully mapped Bengali logic to code (e.g., jodi for if, cholbe for for), lowering the cognitive barrier for native speakers.
2. **Interactive Web IDE:** Created a complete "Windows 95" styled environment where users can manage files, write code with syntax highlighting, and see real-time output.
3. **Dual-Mode Execution:** Achieved the rare feat of supporting both a Virtual Machine execution path and a Native Transpilation path within a single compiler.
4. **Standard Library:** Built a functioning library from scratch that handles String manipulation (strlen, reverse), File I/O (read, write), and Mathematics.

1.9 Technical Accomplishments

From an engineering perspective, the project demonstrates advanced system design:

- **Custom Virtual Machine:** Designed a stack-based VM (vm.c) that executes custom bytecode instructions, managing the instruction pointer and stack frames manually.
- **Garbage Collection:** Implemented a **Mark-and-Sweep Garbage Collector** to automatically manage heap memory, preventing memory leaks during program execution—a complex feature rarely found in student projects.

- **Intermediate Representation (IR):** Successfully implemented a 3-Address Code (3AC) generation phase that decouples the syntax from the execution, allowing for architecture-independent optimizations.
- **Optimization Engine:** Built an optimizer that passes over the IR to simplify algebraic expressions (e.g., converting x * 2 to bitwise shifts) before code generation.

1.10 Identified Limitations

Despite its success, the current version has specific constraints:

- **Scope Resolution:** Variables declared inside nested blocks (like if statements) sometimes exhibit scoping issues in complex recursive scenarios.
- **Debugging Tools:** There is no step-by-step debugger; developers must rely on print statements (dekhaw) to troubleshoot their code.
- **Standard Library Depth:** While functional, the library lacks advanced data structures like HashMaps, Sets, or Linked Lists.

1.11 Long-Term Impact

The long-term impact of Kotha lies in its potential to transform computer science education in Bangladesh. By removing the "English Syntax Barrier," it allows younger students to grasp logic and algorithms years earlier than they typically would. Furthermore, as an open-source project, it serves as a robust educational template for other students to learn how compilers, virtual machines, and garbage collectors work under the hood, fostering a deeper generation of systems engineers.

Chapter 2

Background and Literature Review

This chapter provides the theoretical foundation required for the design and implementation of the compiler. It covers the fundamental concepts of formal language theory, automata, and the standard phases of compilation, including lexical, syntax, and semantic analysis. Additionally, it reviews related tools and existing work to establish the research gap that this project aims to address.

2.1 Formal Language and Automata Theory

Formal language theory is the mathematical backbone of compiler design, providing the tools to define programming languages precisely.

2.1.1 Alphabets, Strings, and Languages

An **alphabet** (Σ) is a finite, non-empty set of symbols (e.g., $\{0, 1\}$ for binary or the set of Unicode characters for a Bangla-based language). A **string** is a finite sequence of symbols chosen from some alphabet. The empty string is denoted by ϵ. A **language** is a set of strings formed over an alphabet. In the context of this project, the "language" is the set of all valid programs that can be written in the custom Banglish syntax.

2.1.2 Regular Expressions

Regular expressions (Regex) are algebraic notations used to describe patterns in strings. They are primarily used to define the **lexical structure** of a programming language, such as identifiers, keywords (e.g., *dhoro*, *jodi*), and literals. A regular expression over an alphabet Σ can describe languages formed by concatenation, union, and Kleene closure operations.

2.1.3 Context-Free Grammars (CFG)

While regular expressions can describe simple patterns, they cannot define the nested structures typical of programming languages (e.g., matching parentheses or nested if-else blocks). **Context-Free Grammars (CFG)** are used to specify the syntax. A CFG is a 4-tuple $G = (V, T, P, S)$, where:

- V is a set of variables (non-terminals).
- T is a set of terminals (tokens like *ID*, *INT*).
- P is a set of production rules mapping variables to strings of variables and terminals.
- S is the start symbol.

2.1.4 Derivation and Parse Trees

Derivation is the process of replacing non-terminals with their corresponding production rules to generate a string. A **parse tree** is a graphical representation of this derivation, where the root is the start symbol, interior nodes are non-terminals, and leaves are terminals. It depicts the hierarchical structure of the source code.

2.1.5 Ambiguity in Grammar

A grammar is said to be **ambiguous** if there exists a string for which there is more than one distinct parse tree (or leftmost derivation). Ambiguity is problematic for compilers because it implies multiple interpretations for the same code (e.g., the "dangling else" problem). Disambiguating rules, such as operator precedence and associativity, are essential to ensure a unique parse tree.

2.1.6 Left Recursion and Left Factoring

- **Left Recursion:** A grammar is left-recursive if a non-terminal A derives a string starting with itself ($A \to A\alpha$). Top-down parsers cannot handle left recursion as it leads to infinite loops. It must be eliminated by rewriting the grammar.
- **Left Factoring:** This is a transformation used when two productions for the same non-terminal share a common prefix. It defers the decision of which production to use until enough input has been seen, which is crucial for predictive parsing.

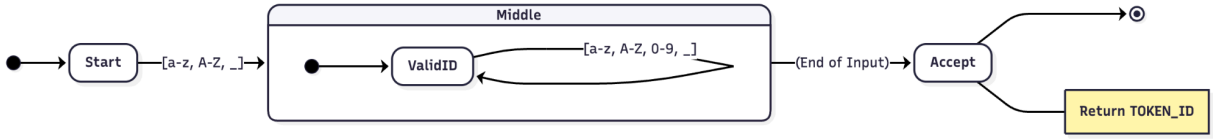
2.2 Lexical Analysis Background

Lexical analysis is the first phase of a compiler, responsible for reading the raw input stream and grouping characters into meaningful sequences.

2.2.1 Token, Lexeme, and Pattern

- **Token:** An abstract category representing a unit of the source code (e.g., TOKEN_VAR, TOKEN_IF).
- **Lexeme:** The actual sequence of characters in the source code that matches the pattern for a token (e.g., the string "dhoro" matching TOKEN_VAR).
- **Pattern:** The rule (usually a regular expression) that describes the set of lexemes representing a specific token.

2.2.2 Finite Automata (DFA/NFA)



Lexical analyzers rely on **Finite Automata** to recognize patterns. A **Nondeterministic Finite Automaton (NFA)** allows multiple transitions for the same input, while a **Deterministic Finite Automaton (DFA)** has exactly one transition for each symbol. Tools like Flex convert regular expressions into an NFA and then optimize it into a DFA for efficient token recognition.

2.2.3 Regular Expressions in Tokenization

In this project, regular expressions are used to define the Banglish keywords and syntax. For example, a regex like [0-9]+ defines an integer, while [a-zA-Z_][a-zA-Z0-9_]* typically defines an identifier.

2.2.4 Role of the Lexer

The lexer's primary role is to strip out whitespace and comments and produce a stream of tokens for the parser. It also tracks line numbers to report lexical errors (e.g., illegal characters)[3333].

2.3 Syntax Analysis Background

Syntax analysis, or parsing, verifies that the string of tokens produced by the lexer conforms to the grammatical rules of the language.

2.3.1 Parsing Concepts

Parsing determines the syntactic structure of the code. It is generally categorized into **Top-Down** (building the tree from the root) and **Bottom-Up** (building the tree from the leaves).

2.3.2 LL(1) and Recursive Descent Parsing

LL(1) is a top-down parsing method that scans input from **L**eft to right, producing a **L**eftmost derivation using **1** lookahead symbol. **Recursive Descent** is a common implementation of top-down parsing where each non-terminal in the grammar has a corresponding function in the code.

2.3.3 Shift-Reduce Parsing

This project utilizes **Shift-Reduce Parsing**, a bottom-up technique used by tools like Bison. It works by "shifting" tokens onto a stack until a pattern matches the right-hand side of a production rule, at which point the tokens are "reduced" to the corresponding non-terminal.

2.3.4 Parse Tree vs. Syntax Tree

- **Parse Tree:** A concrete representation containing all grammar details, including redundant punctuations like parentheses and semicolons.
- **Syntax Tree (AST):** An abstract representation that retains only the essential structural information required for processing (e.g., an operator node with operands as children). The project focuses on generating an AST for the interpreter.

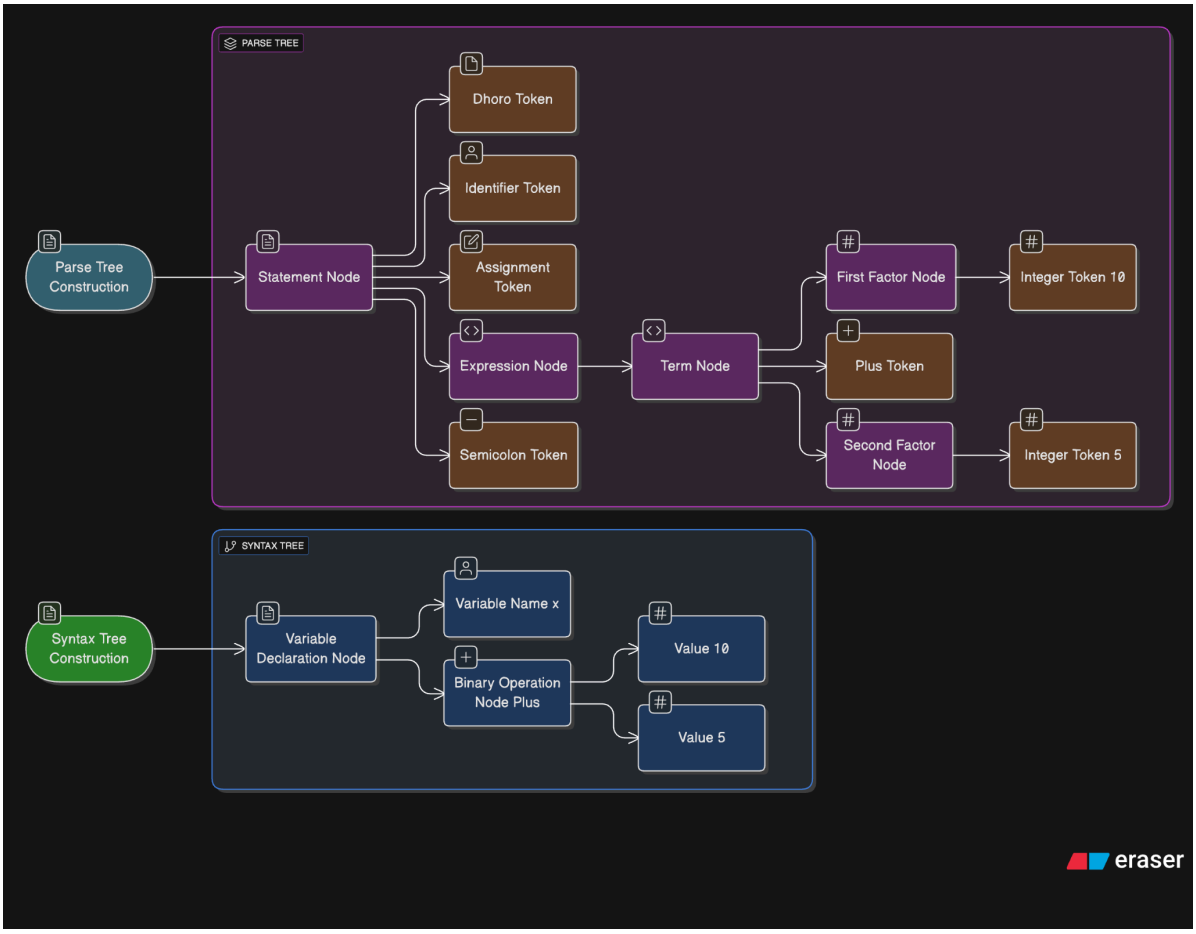


Fig : 2.1

2.3.5 Syntax Errors and Recovery

Compilers must handle errors gracefully. **Panic-mode recovery** involves skipping tokens until a synchronizing token (like a semicolon) is found, allowing the parser to continue checking the rest of the program rather than halting at the first error.

2.4 Semantic Analysis

This phase ensures that the syntactic structures obey the language's semantic rules (meaning).

2.4.1 Symbol Table

The **Symbol Table** is a critical data structure that stores information about identifiers (variables, functions), such as their names, data types, scope, and memory locations. It allows the compiler to verify that a variable like count has been declared before use.

2.4.2 Scope and Type Checking

- **Scope Checking:** Ensures variables are accessible only within their defined regions (e.g., local vs. global).
- **Type Checking:** Verifies that operations are performed on compatible data types (e.g., preventing the division of a string by an integer).

2.4.3 Attribute Grammars

Attribute grammars extend CFGs by associating values (attributes) with grammar symbols. Synthesized attributes pass information up the parse tree, while inherited attributes pass information down, facilitating context-dependent checks.

2.5 Intermediate Code Generation

Before generating target machine code, compilers often generate an intermediate representation (IR) that is machine-independent.

2.5.1 Abstract Syntax Tree (AST)

The AST acts as a high-level IR. It simplifies the program structure, removing syntactic sugar, and serves as the foundation for the interpreter’s execution logic.

2.5.2 Three-Address Code (TAC)

TAC is a linear IR where instructions have at most three operands (e.g., x = y + z). It is easier to optimize and translate into assembly than complex tree structures.

2.6 Code Optimization

Optimization attempts to improve the efficiency of the intermediate code without changing its functionality.

- **Peephole Optimization:** examines a small window of instructions to perform local improvements (e.g., removing redundant loads/stores).
- **Constant Folding:** evaluates constant expressions at compile-time (e.g., replacing 3 + 4 with 7).

- **Dead-Code Elimination:** removes code that is unreachable or whose results are never used, reducing program size.

2.7 Code Generation

The final phase maps the optimized intermediate representation to the target language (assembly or machine code).

- **Instruction Selection:** Choosing the best machine instructions to implement IR operations.
- **Register Allocation:** Assigning variables to a limited number of CPU registers to maximize speed.

2.8 Related Work and Tools

Several standard tools and existing projects form the basis for modern compiler design:

- **Lex and Yacc (Flex and Bison):** The industry-standard tools for generating lexical analyzers and parsers. This project utilizes Flex for tokenizing the Banglish code and Bison for parsing the grammar.
- **GCC and Clang:** Major production-grade compilers for C/C++ that demonstrate robust architecture and optimization pipelines.
- **Interpreters (Python, V8):** Unlike compilers that translate code to machine language, interpreters execute source code directly or via bytecode. This project adopts an interpreter-based approach similar to Python.

2.9 Research Gap and Motivation

While numerous programming languages exist, a significant gap remains in **native-language accessibility** for programming education.

1. **Language Barrier:** Most programming languages rely on English keywords (*if, while, print*), which can be a cognitive barrier for non-native speakers, particularly beginners in Bangladesh.
2. **Lack of Bangla-based Tools:** Existing attempts at Bangla programming languages often lack a robust, full-scale compiler architecture or are merely simple translators.
3. **Motivation:** This project addresses these limitations by creating a **Banglish** (Bengali written in English script) programming language. It is designed to lower the entry barrier for students by allowing them to use familiar logic and vocabulary (e.g., *jodi* for *if*, *ghuraw* for *loop*) while exposing them to the internal workings of a standard compiler architecture using C.

2.10 Summary

This chapter outlined the theoretical frameworks essential for compiler construction, ranging from automata theory to code generation. It established the role of lexical and syntax analysis and highlighted the importance of intermediate representations. Finally, it contextualized the project within the broader field, emphasizing the unique contribution of a Banglish-syntax compiler for educational accessibility.

Chapter 3

Proposed Methodology/Architecture

3.1 Requirement Analysis & Design Specification

The development of a compiler is a rigorous engineering process that requires a systematic transition from abstract language concepts to concrete software implementation. This chapter delineates the proposed methodology for the **Kotha** programming language, detailing the requirement analysis and design specifications. Requirement analysis identifies the functional and non-functional necessities of the system, ensuring the language meets its educational and technical goals. Conversely, the design specification translates these requirements into a modular architectural blueprint, defining the data structures, algorithms, and interfaces required to build a robust compiler and runtime environment.

3.1.1 Overview

Kotha is a statically typed, interpreted programming language designed to bridge the linguistic gap in computer science education. Built using the C programming language, Flex (Lexical Analyzer), and Bison (Parser Generator), Kotha abstracts complex programming concepts into a syntax rooted in the Bengali language (Banglish). The system is not merely a translator but a full-stack compiler that processes source code through a multi-stage pipeline—from lexical analysis to intermediate representation optimization—before executing it on a custom Virtual Machine (VM).

The project encompasses the following core components:

- **The Compiler Core:** Handles tokenization, parsing, semantic analysis, and bytecode generation.
- **The Virtual Machine (VM):** A stack-based runtime environment featuring automatic memory management (Garbage Collection).
- **The Integrated Development Environment (IDE):** A web-based interface providing a seamless coding experience.

Key Features:

- **Localized Syntax:** Use of keywords like dhoro (variable declaration), jodi (conditional), and ghuraw (loop).
- **Control Flow:** Support for if-else branching, while loops, and for loops.
- **Data Types:** Handling of Integers, Floating-point numbers, and Strings.
- **Standard Library:** Built-in support for Math operations, String manipulation, and File I/O.

- **Dual Execution:** Capabilities for both VM execution and native C transpilation.

The educational objective is to demystify compiler construction for students by providing a transparent, open-source implementation that allows learners to observe how high-level code is transformed into machine-executable instructions.

3.1.2 Requirement Analysis & Design Specification

1. Requirement Analysis

a) Functional Requirements

- **Lexical Analysis:** The system must utilize Flex to scan .kotha source files, identifying valid tokens (keywords, identifiers, literals, operators) while discarding whitespace and comments.
- **Syntax Parsing:** The system must utilize Bison to validate the stream of tokens against a Context-Free Grammar (CFG), generating an Abstract Syntax Tree (AST) if the code is syntactically correct.
- **Semantic Analysis:** The compiler must enforce scope resolution (ensuring variables are declared before use) and type safety (preventing incompatible operations).
- **Intermediate Code Generation:** The system must translate the AST into a linear Intermediate Representation (3-Address Code) to facilitate optimization.
- **Optimization:** The compiler must perform basic optimizations such as constant folding and algebraic simplification on the IR.
- **Code Generation:** The system must generate efficient bytecode instructions for the Kotha Virtual Machine.
- **Runtime Execution:** The Virtual Machine must execute the bytecode, managing the operand stack, call frames for functions, and heap memory via Garbage Collection.
- **Error Handling:** The system must detect and report syntax errors and runtime exceptions (e.g., Division by Zero) with informative messages.

b) Non-Functional Requirements

- **Simplicity:** The syntax must be intuitive for Bengali speakers to lower the cognitive load of learning programming.
- **Performance:** The core compiler and VM must be written in C to ensure high execution speed and low memory overhead.
- **Portability:** The system should compile and run on standard Unix/Linux and macOS environments using GCC.
- **Maintainability:** The codebase must adhere to a modular architecture, separating the frontend (analysis) from the backend (synthesis) to facilitate future upgrades.

- **Robustness:** The interpreter must handle invalid inputs gracefully without crashing, ensuring a stable learning environment.

c) Software Requirements

- **Core Language:** C (C99 Standard).
- **Build Tools:** GNU Make, GCC Compiler.
- **Parser Generators:** Flex (Fast Lexical Analyzer Generator), Bison (GNU Parser Generator).
- **IDE Backend:** Python 3 (for the local web server).
- **Development Environment:** Visual Studio Code or any standard text editor.

d) Hardware Requirements

- **Processor:** Minimum 1 GHz Single-core processor (optimized for low-end hardware).
- **RAM:** Minimum 512 MB (Compiler footprint is minimal).
- **Storage:** 100 MB free space for source code, binaries, and temporary build files.

3.2 Architectural Overview

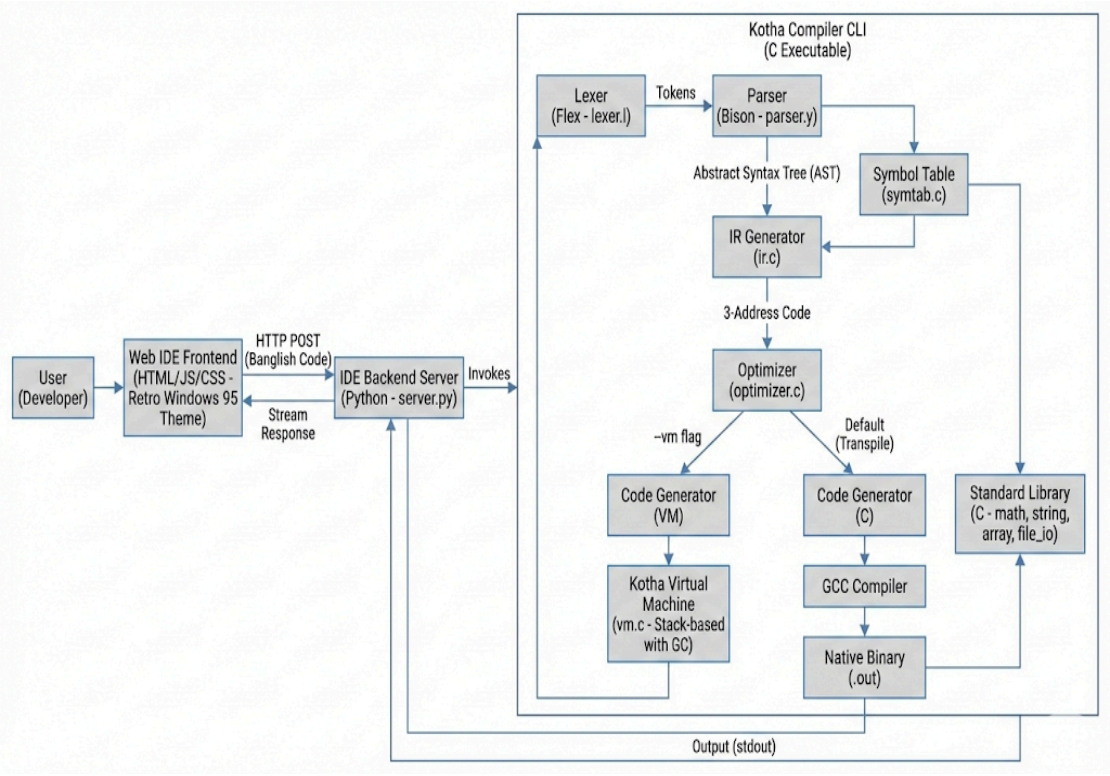


Figure Number 3.1: The projects system architecture.

The Kotha compiler follows a multi-pass architecture comprising the following distinct modules:

1. **Source Input:** The user provides a .kotha file.
2. **Lexer (lexer.l):** Reads characters and produces a stream of tokens.
3. **Parser (parser.y):** Consumes tokens to build the Abstract Syntax Tree (AST).
4. **Semantic Analyzer (symtab.c):** Traverses the AST to populate the Symbol Table and check for logic errors.
5. **IR Generator (ir.c):** Flattens the AST into Quadruples (Intermediate Representation).
6. **Optimizer (optimizer.c):** Refines the IR to remove inefficiencies.
7. **Code Generator (codegen_vm.c):** Converts IR into custom Bytecode.
8. **Virtual Machine (vm.c):** Executes the bytecode, interacting with Standard Libraries (math_lib.c, file_io.c) to produce output.

Data Structures :

Tokens: Defined in the Bison header, representing the smallest units of meaning (e.g., TOKEN_DHORO, TOKEN_INT).

- **Abstract Syntax Tree (AST):** A hierarchical tree structure where nodes represent operators (e.g., NODE_IF, NODE_ASSIGN) and leaves represent operands.
- **Symbol Table:** A Hash Table structure used to store variable names, data types, and memory locations, supporting scope management via chaining.
- **Intermediate Representation (IR):** A doubly linked list of instructions, where each node contains an operator, up to two arguments, and a result destination.
- **Bytecode:** An array of integer-based instructions (OpCodes) designed for the stack-based VM.

Design Principles :

- **Modularity:** Each phase of compilation is isolated in its own C file (lexer.l, parser.y, vm.c), allowing individual components to be tested or replaced without affecting the whole system.
- **Extensibility:** The architecture supports the easy addition of new keywords or library functions by simply registering them in the symbol table and VM instruction set.
- **Error Isolation:** Syntax errors are caught in the parsing phase, while logic errors are caught in the semantic phase, preventing the execution of flawed code.

Clean Pipeline: Data flows strictly in one direction (Analysis $\rightarrow$ Synthesis), ensuring a predictable compilation process.

3.2.1 Interface Design

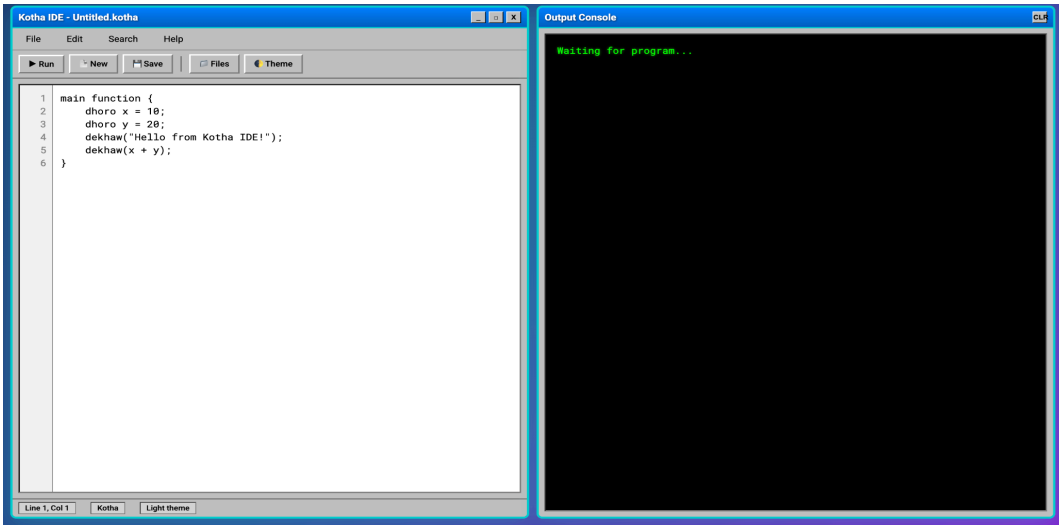


Figure 3.2 : Interface Design Theme Day .

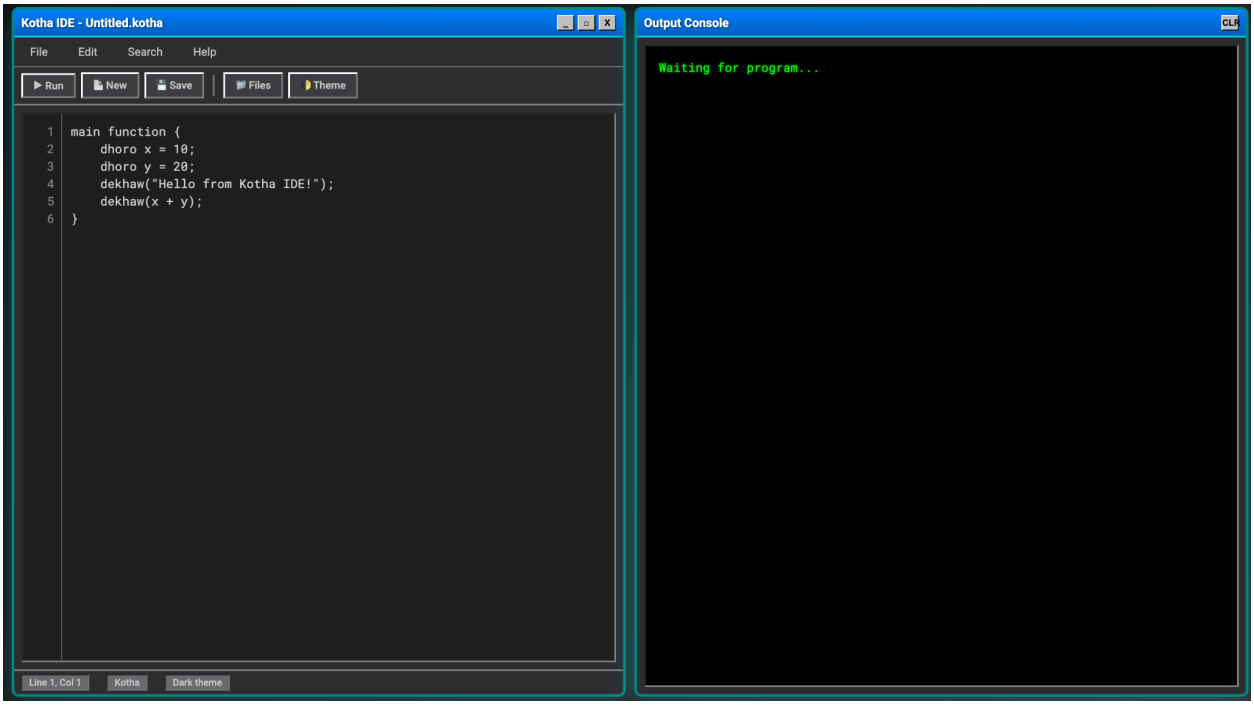


Figure 3.3: Interface Design Theme Night .

While the core of Kotha is a command-line tool, the project includes a web-based **Integrated Development Environment (IDE)** to enhance user accessibility. The UI is designed with a "Retro / Windows 95" aesthetic to provide a distinct visual identity while prioritizing functionality.

Interface Objectives:

- To provide an accessible platform for writing and running Kotha code without complex local installation.
- To visualize the immediate output of the code execution.
- To highlight syntax errors in real-time.

3.2.3User Interaction Flow: Runtime User Input Acquisition Flow

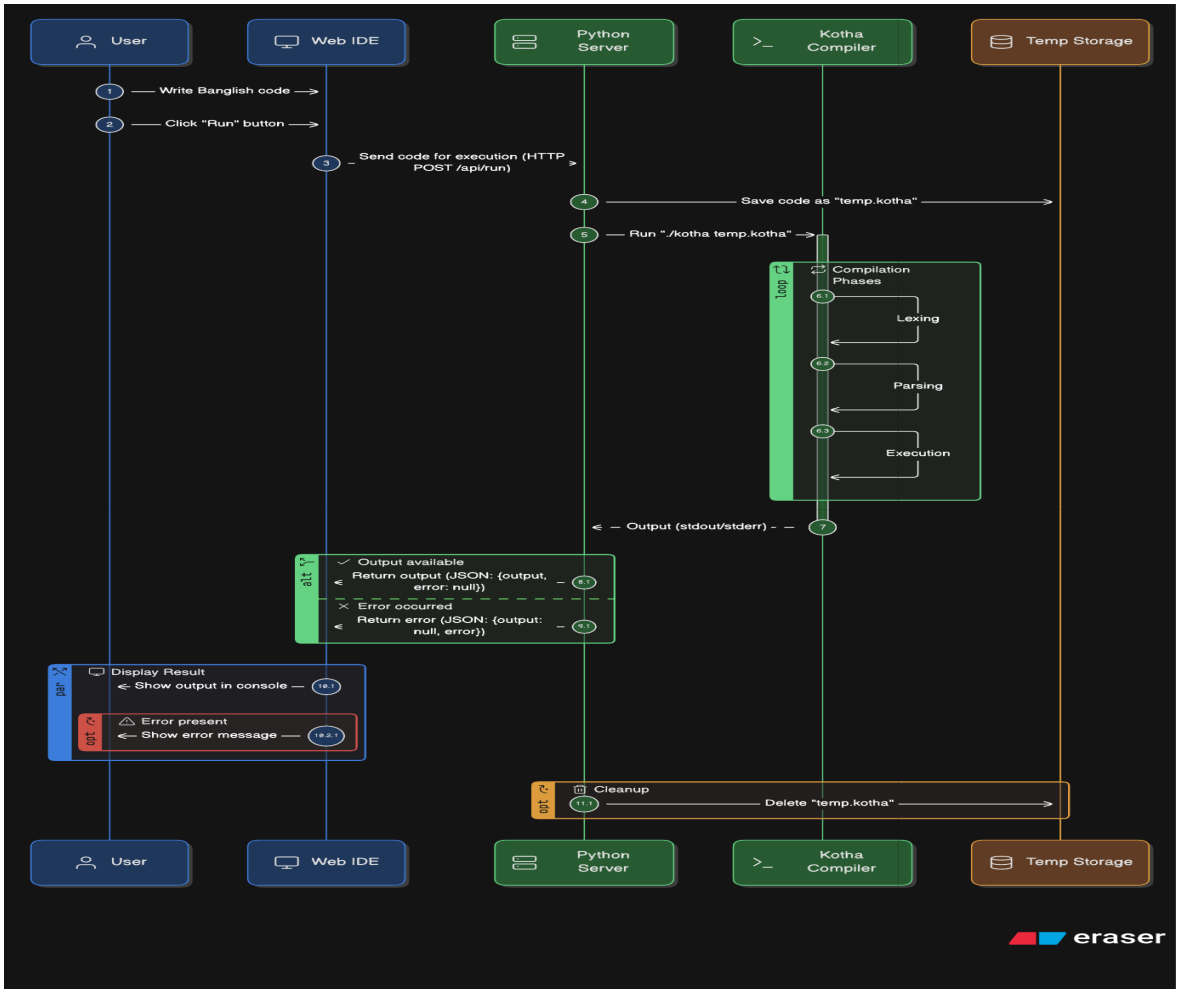


Figure 3.4: User Interaction Flow diagram

1. **Code Writing:** The user types Banglish code into the editor pane (e.g., dekhaw("Hello");).

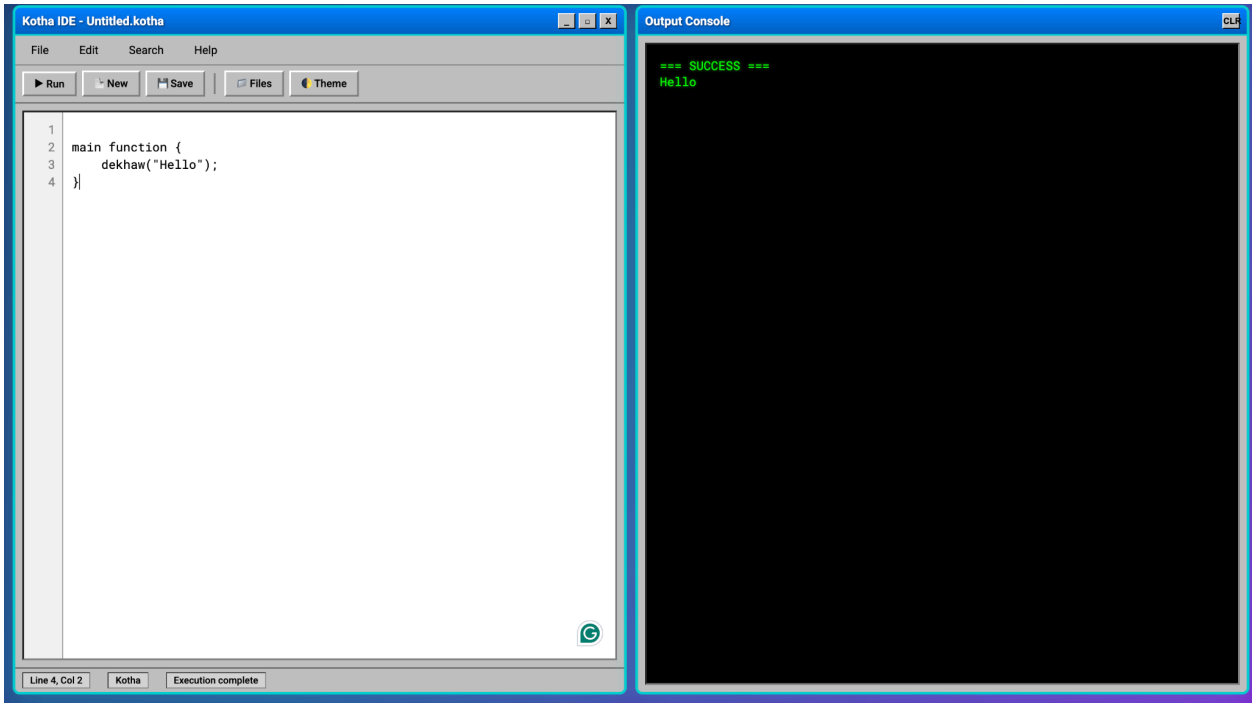


Figure 3.5 : Code writing Dekhao (Hello)

2. **Execution:** Runtime Input Acquisition Flow **.**

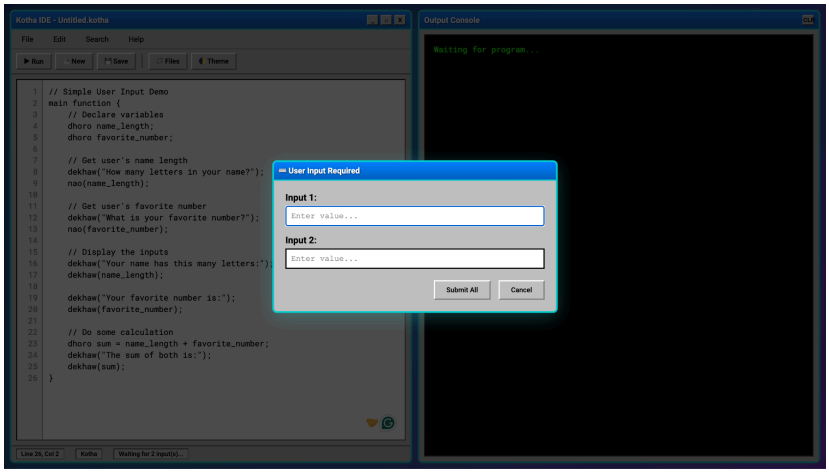


Figure 3.6 : Interactive User Input Dialog Interface

3. **Processing:** The frontend sends the code via an HTTP POST request to the Python backend (server.py).

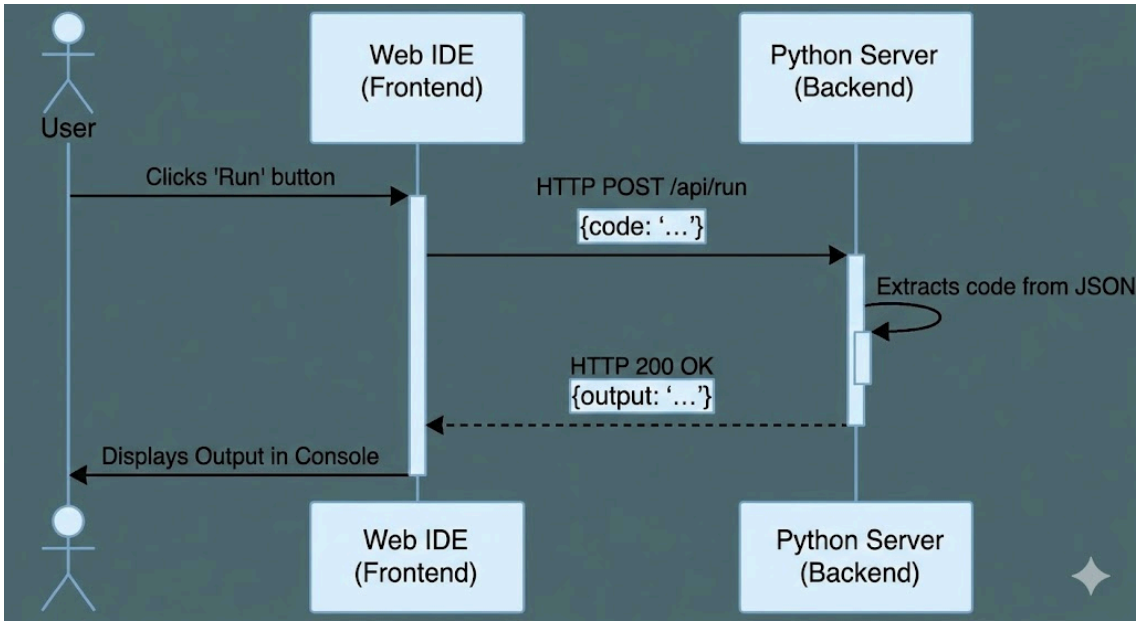


Figure 3.7: The frontend sends the code via an HTTP POST request diagram

4. **Compilation:** The backend saves the code to a temporary file, invokes the Kotha compiler binary, and captures the stdout and stderr.

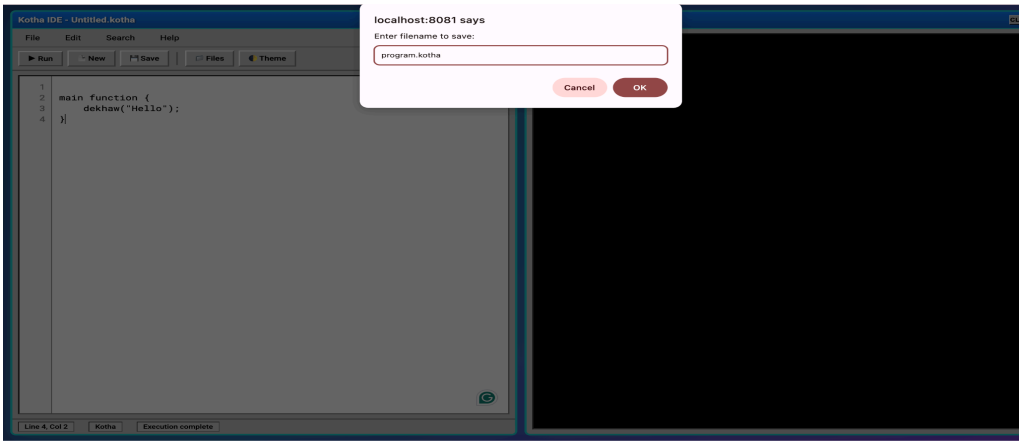


Figure 3.8 :The backend saves the code to a temporary file diagram

5. **Help Button :** IDE Help Interface & Command Reference

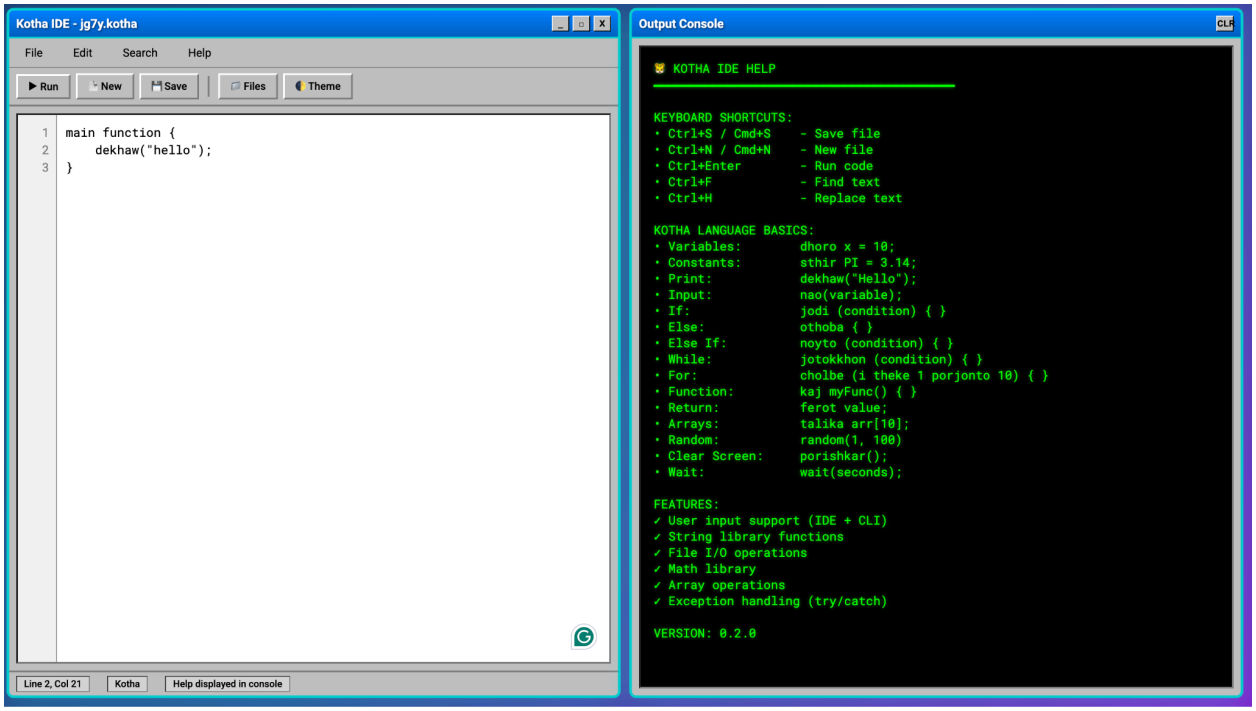


Figure 3.9 : User Support & Documentation System

6) File Button : File Management Interface & Options

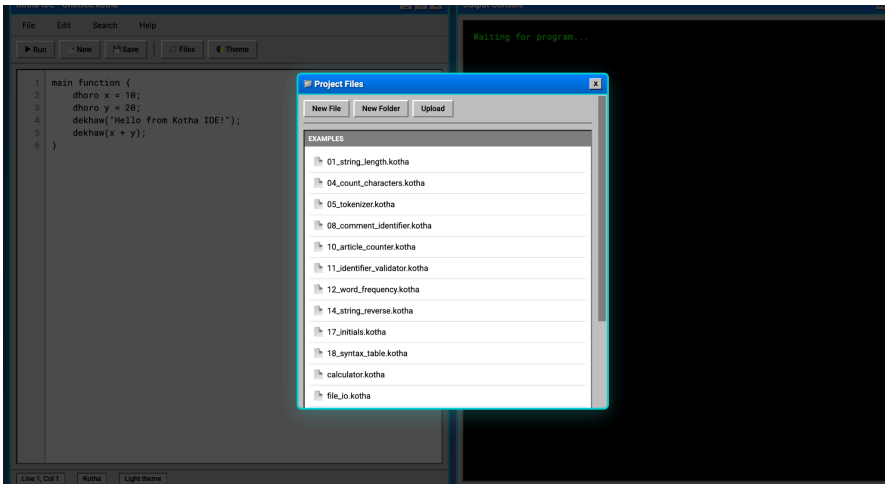


Figure 3.10 : Project File Operations Menu Diagram

Error Reporting:

The UI distinguishes between two types of errors:

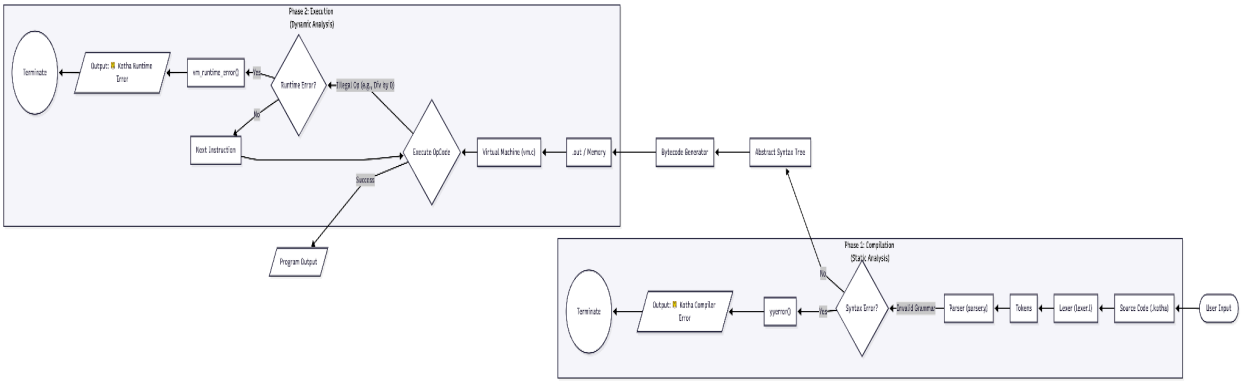


Figure 3.11 : Error Detection and Handling Architecture

This diagram illustrates the two distinct stages where errors are intercepted in the Kotha system:

1. **Syntax Errors (Compile Time):**
    - **Occurrence:** Occurs during the Parsing phase when the code violates the grammatical rules defined in parser.y (e.g., missing semicolon, unbalanced braces).
    - **Mechanism:** The **Bison** parser detects the mismatch and triggers the yyerror() function.
    - **Output:** The compiler immediately halts and prints a formatted message: Kotha Compiler Error with the line number.
2. **Runtime Errors (Execution Time):**
    - **Occurrence:** Occurs inside the **Virtual Machine** (vm.c) when the code is syntactically correct but performs an illegal operation (e.g., Division by Zero, Stack Overflow).
    - **Mechanism:** Before executing specific bytecodes (like OP_DIV), the VM checks validity (e.g., is the denominator 0?). If invalid, it calls vm_runtime_error().
    - **Output:** The program crashes gracefully with a stack trace and the message: Kotha Runtime Error.

3.3 Overall Project Plan

The project was executed following a structured plan that ensured systematic development, testing, and delivery of a working minimalist interpreted programming language. The plan was designed to balance technical implementation with educational objectives.

Project Goals:

- To design a formal grammar for a Bengali-syntax programming language.
- To implement the full compiler pipeline (Lexer, Parser, IR, CodeGen).
- To build a custom Virtual Machine with Garbage Collection.
- To develop a Standard Library for file and math operations.
- To create a user-friendly Web IDE.

Development Phases:

Phase 1 – Requirement Analysis & Planning

- Defined the scope of the "Banglish" syntax.
- Selected the technology stack (C, Flex, Bison).
- Drafted the Context-Free Grammar (CFG) rules.

Phase 2 – System Design

- Designed the modular architecture: Source $\rightarrow$ AST $\rightarrow$ IR $\rightarrow$ VM.
- Defined core data structures (AST Nodes, Symbol Table, Stack Frames).
- Designed the Virtual Machine instruction set architecture.

Phase 3 – Implementation

- **Frontend:** Implemented lexer.l and parser.y.
- **Middle-end:** Developed ast.c and symtab.c for semantic analysis.
- **Backend:** Built the vm.c and the Bytecode Generator.
- **Libraries:** Implemented math_lib.c, string_lib.c, and file_io.c.

Phase 4 – Testing & Debugging

- Created unit tests for individual modules (Array, Math).
- Developed a suite of .kotha example programs (Factorial, Fibonacci).
- Debugged memory leaks in the Garbage Collector.
- Verified the integration between the Python backend and the C compiler.

Phase 5 – Documentation & Finalization

- Documented the language syntax and usage guide.
- Prepared the comprehensive project report.
- Finalized the IDE UI and deployment scripts.

Table 3.4: Project Timeline and Work Breakdown

Week	Task
Week 1	Requirement gathering, Technology selection, Grammar design.
Week 2	Implementation of Lexical Analyzer (Flex) and Token definitions.
Week 3	Implementation of Parser (Bison) and Abstract Syntax Tree.
Week 4	Symbol Table integration and Semantic Analysis.
Week 5	Development of Intermediate Representation and Optimizer.
Week 6	Construction of the Virtual Machine and Bytecode Generator.
Week 7	Implementation of Standard Libraries and Garbage Collector.
Week 8	Web IDE development, Integration testing, and Documentation.

3.4 Summary of the Chapter

This chapter provided a detailed exposition of the methodology and architecture employed in the development of the Kotha programming language. It began with a rigorous requirement analysis, establishing the functional needs for a lexer, parser, and virtual machine, alongside non-functional goals like performance and maintainability. The design specification outlined the modular system architecture, detailing the data structures and interaction flow between the compiler's frontend and backend. Finally, the project plan demonstrated a structured, iterative approach to development, ensuring that each phase—from conceptual design to final testing—contributed to a robust and educational software system. The subsequent chapters will detail the specific implementation of these modules.

Chapter 4

Method

This chapter details the technical methodology and core components utilized in the construction of the Kotha programming language. It presents a comprehensive overview of the language's design, including the core syntax, keyword specifications, and the standard library functions implemented for file I/O, string manipulation, and mathematics. Additionally, it describes the development tools and the execution environment, supported by code excerpts and visual representations that illustrate the practical application of these methods.

4.1 Core Language

Banglish Syntax: Familiar keywords like purno (int), doshomik (float), bornona (string), dekhaw (print), jodi (if), noyto (else), jotokkhon (while).

Data Types: Support for Integers (purno), Floats (doshomik), Strings (bornona), and Booleans (sotyo_mittha).

C-Style Declarations: Multiple variable declarations in one line: purno x, y, z = 10;

String Concatenation: Use the + operator to concatenate strings with automatic type conversion: "Age: " + age

Type Casting: 12 conversion functions for explicit type conversions (e.g., purno_to_doshomik(), bornona_to_purno())

Type Inspection: typeof() operator for runtime type checking

Control Flow: if-else conditionals and while loops.

Functions: Define and call functions with arguments and return values.

Virtual Machine: Code is compiled to bytecode and executed on a custom high-performance VM.

4.2 Standard Library

String Library: String manipulation functions such as length, concatenate, reverse, etc.

Math Library: Common mathematical functions including sin, cos, power, sqrt.

File I/O: Read from and write to files directly from Kotha.

Array Support: Create and manipulate arrays.

4.3 Development Tools

Kotha IDE: A web-based Integrated Development Environment with syntax highlighting and file management.

REPL: interactive Read-Eval-Print Loop for quick coding sessions.

CLI: Command-line interface for compiling and running scripts.

4.4 Keyword Reference

Complete list of all Kotha keywords organized by category:

4.4.1 Data Types

Keyword	English	Description	Example
purno	Integer	Whole numbers	purno age = 25;
doshomik	Float	Decimal numbers	doshomik pi = 3.14;
bornona	String	Text/characters	bornona name = "Kotha";
sotyo_mittha	Boolean	True/False values	sotyo_mittha flag = sotti;
sthir	Constant	Immutable value	sthir purno MAX = 100;

4.4.2 Boolean Literals

Keyword	English	Description
sotti	TRUE	Boolean true value
mittha	FALSE	Boolean false value

4.4.3 Control Flow

Keyword	English	Description	Example
jodi	If	Conditional statement	jodi (x > 0) { ... }
noyto	Else	Alternative condition	noyto { ... }
othoba	Else If	Chained condition	othoba (x < 0) { ... }
jotokkhon	While	Loop while condition is true	jotokkhon (i < 10) { ... }
cholbe	For	For loop	cholbe (i theke 0 porjonto 10) { ... }
theke	From	Loop start value	Used with cholbe
porjonto	To	Loop end value	Used with cholbe
paltaw	Switch	Switch statement	paltaw (x) { ... }
holo	Case	Switch case	holo 1: { ... }
ses	Break	Exit loop/switch	ses;

4.4.4 Functions :

Keyword	English	Description	Example
kaj	Function	Define a function	add kaj (purno a, purno b) { ... }
ferot	Return	Return value from function	ferot a + b;
void	Void	Function with no return value	print kaj void () { ... }
main function	Main	Program entry point	main function { ... }

4.4.5 Input/Output :

Keyword	English	Description	Example
dekhaw	Print	Output to console	dekhaw("Hello");
nao	Input	Read user input	nao(x);

4.4.6 Type Operations :

Keyword	English	Description	Example
typeof	Type Of	Get variable type	typeof(x) returns "purno"

4.4.7 Type Conversion Functions :

Function	Description	Example
purno_to_doshomik()	Int → Float	purno_to_doshomik(42) → 42.0
purno_to_bornona()	Int → String	purno_to_bornona(42) → "42"
purno_to_sotyo_mittha()	Int → Boolean	purno_to_sotyo_mittha(1) → sotti
doshomik_to_purno()	Float → Int	doshomik_to_purno(3.14) → 3
doshomik_to_bornona()	Float → String	doshomik_to_bornona(3.14) → "3.14"
doshomik_to_sotyo_mittha()	Float → Boolean	doshomik_to_sotyo_mittha(0.0) → mittha
bornona_to_purno()	String → Int	bornona_to_purno("42") → 42
bornona_to_doshomik()	String → Float	bornona_to_doshomik("3.14") → 3.14
bornona_to_sotyo_mittha()	String → Boolean	bornona_to_sotyo_mittha("") → mittha
sotyo_mittha_to_purno()	Boolean → Int	sotyo_mittha_to_purno(sotti) → 1
sotyo_mittha_to_doshomik()	Boolean → Float	sotyo_mittha_to_doshomik(sotti) → 1.0
sotyo_mittha_to_bornona()	Boolean → String	sotyo_mittha_to_bornona(sotti) → "sotti"

4.4.8 Data Structures :

Keyword	English	Description	Example
talika	Array	Array/List	purno talika arr = new talika[10];
new	New	Create new object/array	new talika[size]

4.4.9 Utility :

Keyword	English	Description	Example
random	Random	Generate random number	purno r = random();
porishkar	Clear	Clear screen	porishkar();
wait	Wait	Pause execution	wait(1000);
songjukto	Concatenate	String concatenation	songjukto(str1, str2)

4.4.10 Exception Handling :

Keyword	English	Description	Example
try	Try	Try block	try { ... }
catch	Catch	Catch exceptions	catch (error) { ... }
finally	Finally	Finally block	finally { ... }
throw	Throw	Throw exception	throw error;

4.4.11 Operators :

Operator	Description	Example
#ERROR!	Addition / String concatenation	x + y or "Hello " + name
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulo	x % y
#ERROR!	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal	x >= y
<=	Less than or equal	x <= y
&&	Logical AND	x && y
!	Logical NOT	!x

4.5 Specific Notes

Windows Users:

If using WSL, you get the full Linux experience with better compatibility

MinGW/MSYS2 works but may require adjusting file paths in scripts

The IDE server requires Python 3.x - ensure it's in your PATH

Use forward slashes (/) in file paths, not backslashes (\)

macOS Users:

Xcode Command Line Tools include GCC/Clang

Install Homebrew for easy package management: brew install flex bison

Linux Users:

Install build tools: sudo apt install build-essential flex bison (Debian/Ubuntu)

Or: sudo yum install gcc make flex bison (RHEL/CentOS)

4.6 Project Structure

```
Kotha0.2/
├── assets/                 # Images for documentation
├── kotha/                  # Core compiler source code
│   ├── kotha               # Compiled executable
│   ├── parser.y            # Bison parser grammar
│   ├── lexer.l             # Flex lexer definitions
│   ├── vm.c                # Virtual Machine implementation
│   ├── ir.c                # Intermediate Representation generation
│   ├── codegen_vm.c        # Bytecode generation
│   └── *_lib.c             # Standard libraries (math, string, file, array)
├── kotha-ide/              # Web-based IDE
│   ├── examples/           # Sample Kotha programs
│   ├── public/             # Frontend assets (HTML, CSS, JS)
│   └── server.py           # Backend server (Python)
├── PROJECT_REPORT.md       # Detailed technical documentation
├── README.md               # Main documentation
└── run_kotha.sh            # Helper script to execute programs
```

4.7 Examples Code

4.7.1 Hello World Print :

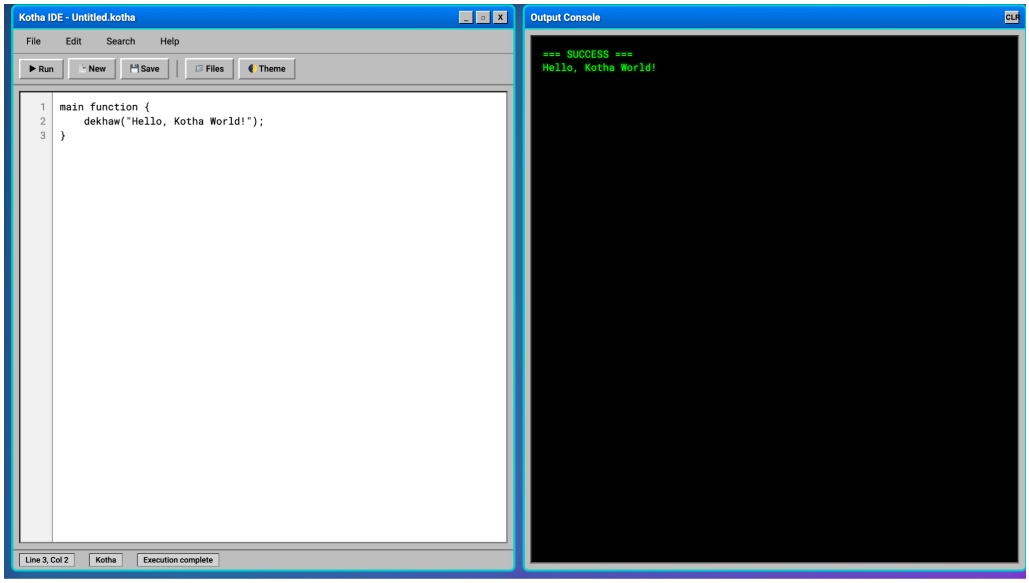


Figure 4.7.1: Print hello world

4.7.2 Variables & Type Casting :

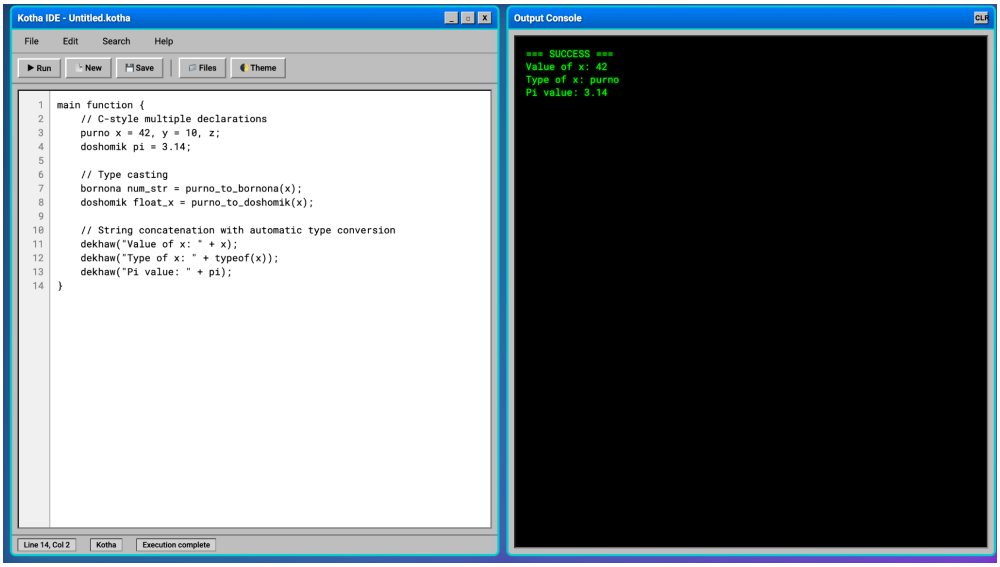


Figure 4.7.2 :Variables & Type Casting

4.7.3 Control Flow :

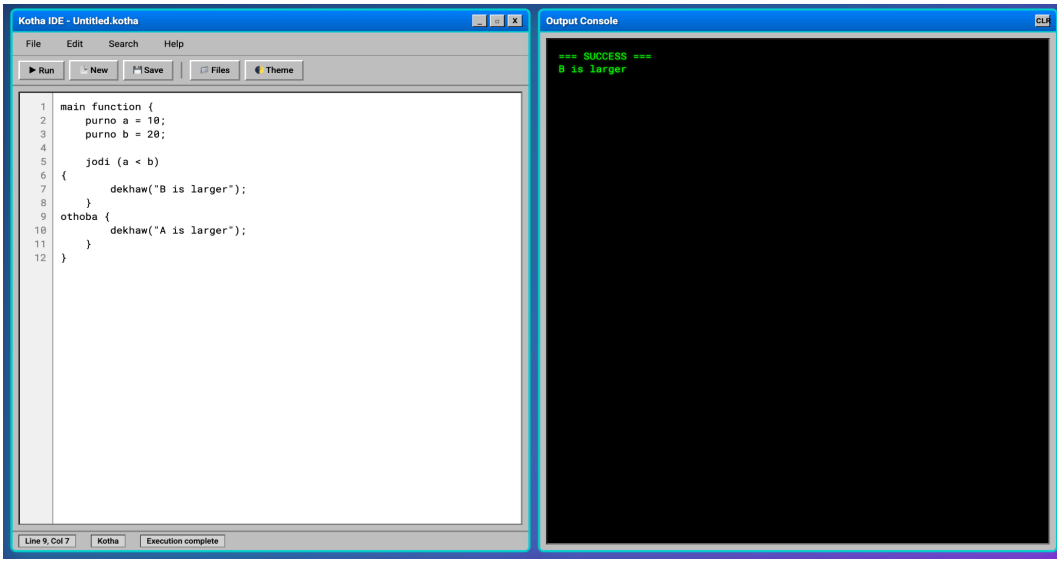


Figure 4.7.3:Control Flow

4.7.4 String Manipulation :

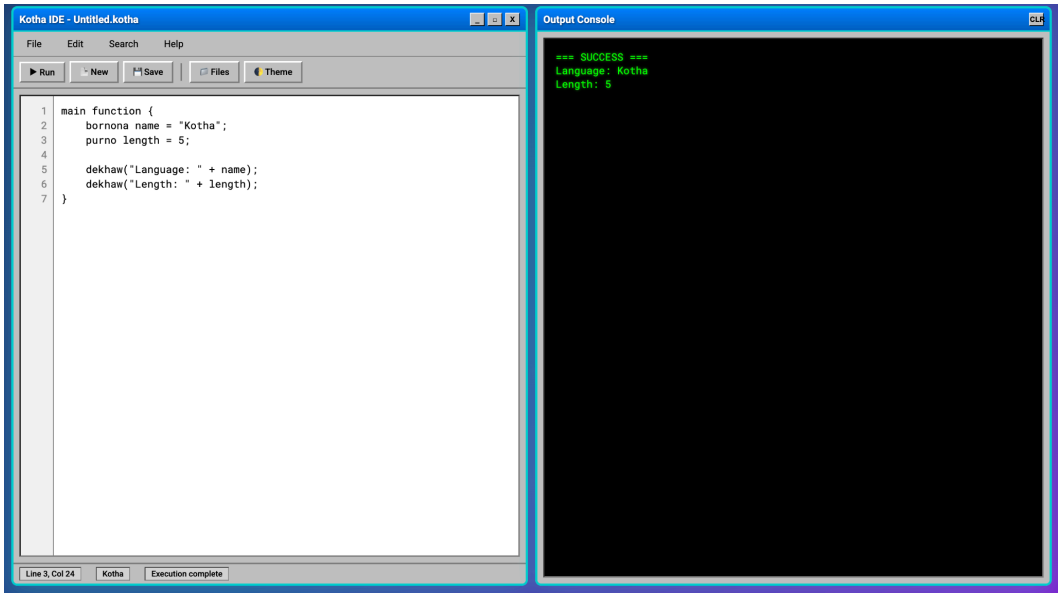


Figure 4.7.4 : String Manipulation

4.7.5 Loops :

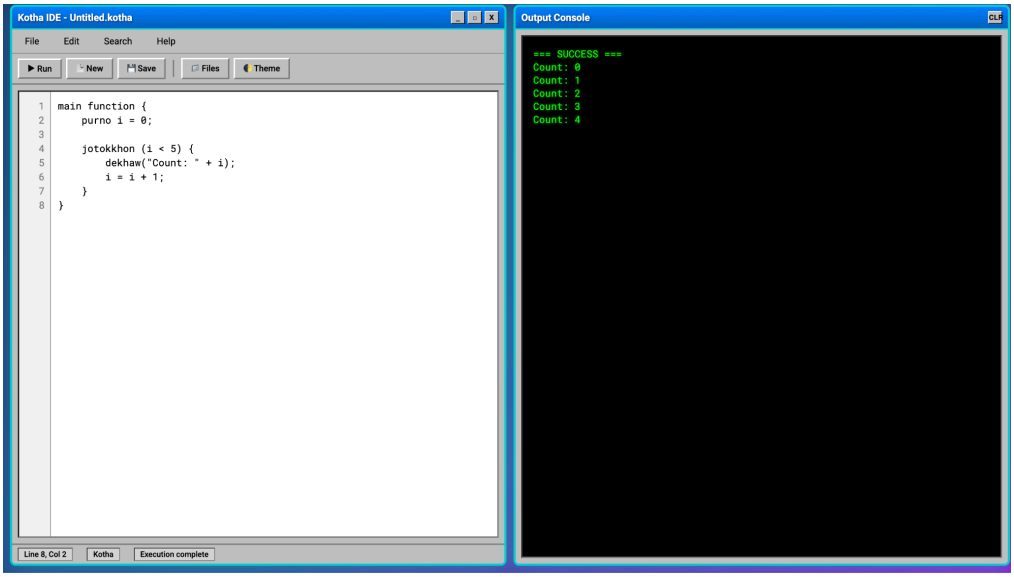


Figure 4.7.5 : Loops

4.7.6 Functions :

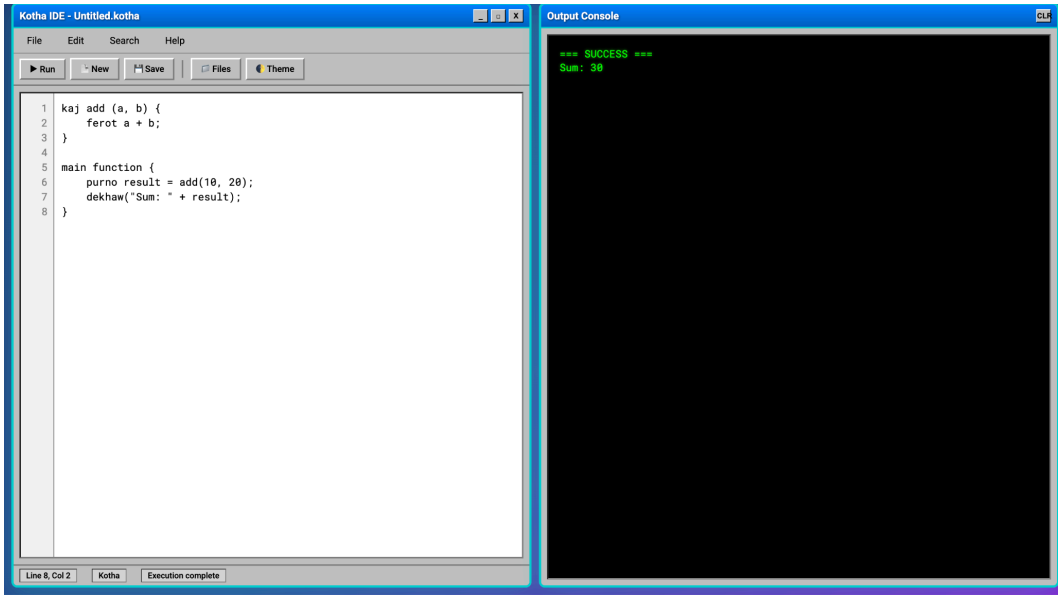


Figure 4.7.7 : Functions

Chapter 5

Engineering Standards and Mapping

This chapter evaluates the development of the **Kotha** programming language against established software engineering standards. It maps the project outcomes to complex engineering problems (CEP) and activities, demonstrating how the compiler adheres to rigorous design principles, ethical considerations, and sustainability goals.

5.1 Impact on Society, Environment & Sustainability

The development of Kotha extends beyond technical achievement; it addresses a significant socio-technical gap in the context of Bangladesh.

5.1.1 Impact on Life

Language barriers act as a major deterrent for students entering the field of Computer Science. By enabling students to write code in "Banglish" (e.g., jodi, ghuraw, dekhaw), Kotha directly impacts the educational lives of native Bengali speakers. It lowers the cognitive load required to understand programming logic, allowing younger students and those from rural backgrounds to acquire computational thinking skills earlier in their education.

5.1.2 Impact on Society & Environment

- **Societal Impact:** Kotha fosters digital inclusivity. It promotes the idea that technology can be adapted to local cultures, encouraging a new generation of engineers who feel represented in the tools they use.
- **Environmental Impact:** Unlike heavy, interpreted languages that require substantial runtime environments (like the JVM or Electron), the Kotha compiler and Virtual Machine are written in optimized **C**. This results in a lightweight binary (~200KB) that consumes minimal CPU and RAM. This efficiency contributes to "Green Computing" by allowing the language to run smoothly on older, low-spec hardware often found in educational labs, reducing electronic waste.

5.1.3 Ethical Aspects

- **Open Source & Transparency:** The project is released as open-source software. The source code is transparent, allowing educators to inspect the inner workings (Lexer, VM) for teaching purposes without hidden proprietary logic.
- **Data Privacy:** Kotha operates entirely offline. The compiler does not collect user data, telemetry, or source code, ensuring complete privacy and integrity for the users.

5.1.4 Sustainability Plan

To ensure the project remains viable long-term:

1. **Modular Architecture:** The system separates the Frontend (parser.y) from the Backend (vm.c), allowing future developers to upgrade individual components without rewriting the core.
2. **Documentation:** Extensive documentation, including a "How to Contribute" guide and architecture diagrams, facilitates community maintenance.
3. **Standardization:** By adhering to C99 standards, the compiler remains compatible with modern operating systems (Linux, macOS, Windows) for the foreseeable future.

5.2 Complex Engineering Problem

This section demonstrates how the project addresses complex engineering challenges through specific mappings to Program Outcomes (POs) and ABET/BAETE criteria.

5.2.1 Mapping to Program Outcomes (POs)

Table 5.1 — Justification of Program Outcomes (POs)

PO	Justification
PO1 (Engineering Knowledge)	Applied foundational knowledge of **Automata Theory**, **Formal Grammars**, and **Data Structures** (Hash Maps, Trees) to build the compiler components.
PO2 (Problem Analysis)	Analyzed the linguistic barriers in programming and formulated a solution using a custom syntax parser and intermediate representation.
PO3 (Design/Development)	Designed a complete system architecture including an **Abstract Syntax Tree (AST)** and a **Virtual Machine (VM)** with custom opcodes.

PO4 (Investigation)	Investigated different parsing strategies (LL vs LALR) and memory management techniques to implement the Garbage Collector.
PO5 (Modern Tool Usage)	Utilized industry-standard tools: **Flex** (Lexical Analysis), **Bison** (Parsing), **GCC** (Compilation), and **Python** (IDE Backend).

5.2.2 Complex Problem Solving

Table 5.2 — Mapping with Complex Problem Solving

EP	Metric	Project-Specific Explanation
EP1	**Depth of knowledge**	Requires deep understanding of **Compiler Construction**, including NFA/DFA conversion for the Lexer and Shift-Reduce parsing logic for the Grammar.
EP2	**Range of conflicting requirements**	Balanced the trade-off between **Language Simplicity** (for beginners) and **Execution Performance** (optimizing the VM and Bytecode generation).
EP3	**Depth of analysis**	Analyzed and resolved complex **Grammar Ambiguities** (e.g., "Dangling Else" problem) using precedence rules in Bison.

EP4	**Familiarity of issues**	Addressed the localized problem of **Language Barriers** in technical education, a specific issue relevant to the Bangladeshi context.
EP5	**Extent of applicable codes**	Adhered to the **C99 Standard** for the core implementation and strict BNF notation for grammar definition.
EP6	**Extent of stakeholder involvement**	Designed the syntax based on feedback regarding readability for Bengali students (the primary stakeholders).
EP7	**Interdependence**	The Frontend (Lexer/Parser) and Backend (VM) are tightly interdependent; changes in the Grammar require updates to the Code Generator.

5.2.3 Complex Engineering Activities

Table 5.3 — Mapping with Complex Engineering Activities

Activity	Project Examples
EA1 — Documentation & information management	Created comprehensive technical documentation including grammar specifications, user manuals, and architectural diagrams.

EA2 — Communication & presentation	Presented the **System Architecture** and **Data Flow** (AST → IR → VM) to explain internal logic during project reviews.
EA3 — Innovation	Innovated by creating a **Dual-Mode Compiler** that supports both VM execution and native C-transpilation, a feature rare in student projects.
EA4 — Investigation & experimentation	Experimented with **Garbage Collection** algorithms (Mark-and-Sweep) to solve memory leaks in the runtime environment.
EA5 — Risk, ethics & sustainability	Mitigated the risk of system crashes through rigorous error handling and designed the tool to be free and open-source for sustainability.

5.3 Engineering Standards Applied

The project rigorously adhered to specific engineering standards to ensure reliability, security, and maintainability.

5.3.1 Coding & Design Standards

- **Language Standards:** The core system is written in **C** following the **ISO/IEC 9899:1999 (C99)** standard, ensuring portability across platforms.
- **Modular Design:** The Codebase follows the **Separation of Concerns** principle. The Lexer (lexer.l), Parser (parser.y), and VM (vm.c) are isolated modules, communicating via defined interfaces (headers).
- **Naming Conventions:** Consistent naming (e.g., node_create, vm_push) was used to maintain readability.

5.3.2 Testing & Verification

- **Unit Testing:** Individual components like the **Math Library** (math_lib.c) and **String Library** were tested with specific test cases (e.g., test_factorial.kotha).

- **Integration Testing:** The full pipeline was verified by running complex scripts (e.g., Fibonacci sequence) and comparing the output against expected results.
- **Feature Mapping:** A feature-to-outcome table was used to verify that every implemented feature met a specific course outcome.

Table 5.4 — Feature-to-Outcomes Mapping

Feature / Activity	Course Outcome
Lexer & Regex Definitions	Demonstrate lexing, regular languages, and finite automata.
Parser (Bison Generated)	Apply parsing algorithms; distinguish LL vs LR properties.
AST & Semantic Checks	Perform type checking, scope management, and semantic analysis.
IR Design & Optimization	Demonstrate intermediate representations and basic optimizations.
Code Generation (VM)	Translate high-level constructs into lower-level code.
Test-Suite & Grading	Apply software testing principles and reproducible workflows.

5.3.3 Security & Safe Execution

- **Memory Safety:** To prevent segmentation faults and memory leaks common in C programs, a **Garbage Collector** was implemented to automatically reclaim unused memory.

- **Bounds Checking:** The Virtual Machine includes strict bounds checking on the Stack and Heap to prevent buffer overflow attacks during execution.
- **Sandboxing:** The Web IDE runs the compiler in a controlled backend environment, preventing user code from affecting the host server's integrity.

5.4 Project Management and Team Work

This chapter outlines the managerial aspects of the **Kotha** compiler project. It details the development methodology, the schedule, cost and risk analysis, the mapping of features to outcomes, and the distribution of roles among team members.

5.4.1 Project Management Methodology

The project followed an **Iterative and Incremental** development model. Instead of attempting to build the entire compiler at once, the development was broken down into functional modules (Frontend, Backend, Runtime).

- **Agile Approach:** The project was divided into small "sprints." For example, the first sprint focused solely on tokenizing inputs (Lexer), while subsequent sprints handled parsing and execution.
- **Version Control: Git** was used for source code management. A centralized repository on **GitHub** hosted the codebase, allowing for incremental commits and feature branching.

5.4.2 Project Schedule

The development timeline was structured over a period of 8 weeks.

Table 5.5 — Project Development Timeline

Timeframe	Phase	Key Activities & Milestones
Week 1	**Requirement Analysis**	• Defined "Banglish" syntax rules. • Selected tools (Flex, Bison, C).
Week 2	**Frontend Development**	• Implemented the **Lexer** to recognize tokens.

		• Designed the **Symbol Table**.
Week 3	**Parser Implementation**	• Developed the **Parser** using Bison. • Created the **AST** structures.
Week 4	**Semantic Analysis**	• Integrated Type Checking. • Implemented Scope Management logic.
Week 5	**Backend Engineering**	• Built the **IR Generator**. • Implemented the **Virtual Machine (VM)**.
Week 6	**Runtime & GC**	• Developed the **Garbage Collector**. • Implemented Standard Libraries.
Week 7	**IDE Integration**	• Built the Python Backend & Web Frontend. • Integrated Syntax Highlighting.
Week 8	**Testing & Reporting**	• Conducted unit testing. • Finalized Project Report.

5.4.3 Cost Analysis

Although this is an academic project built with open-source tools, a theoretical cost analysis was conducted to estimate the resources required for a real-world deployment.

Table 5.6 — Estimated Project Cost

Resource Category	Details	Estimated Cost
Development Tools	GCC, Flex, Bison, VS Code	**$0 (Open Source)**
Hosting	Python Server / Render Free Tier	**$0**
Manpower	3 Developers × 150 Hours	**Academic Credit**
Hardware	Use of personal laptops	**N/A**
Total		**$0**

5.4.4 Risk Management

Potential risks were identified early, and mitigation strategies were put in place.

Table 5.7 — Risk Assessment and Mitigation

Risk Factor	Impact	Mitigation Strategy
Parser Conflicts	High (Syntax errors)	Explicitly defined operator precedence in Bison to resolve shift/reduce conflicts.

Memory Leaks	Critical (System crash)	Implemented a **Mark-and-Sweep Garbage Collector** to manage heap memory automatically.
Scope Creep	Medium (Delay)	Stuck to a strict MVP (Minimum Viable Product) feature set, postponing "Classes" and "ML" to future versions.

5.4.5 Feature-to-Outcomes Mapping

This section maps the specific technical features of **Kotha** to the intended learning outcomes of the Compiler Design course.

Table 5.8 — Feature-to-Outcomes Mapping

Feature / Activity	Course Outcome Demonstrated
Lexer (lexer.l) & Regex Definitions	Demonstrate understanding of **lexical analysis**, regular expressions, and finite automata (DFA/NFA).
Parser (parser.y) & Grammar	Apply **parsing algorithms** (LALR) and distinguish between LL and LR properties using Bison.
AST & Semantic Checks	Perform **type checking**, scope management, and semantic analysis using the Symbol Table.
IR Design & Optimization	Demonstrate **intermediate representations** (3-Address Code) and basic optimizations like constant folding.
Code Generation (VM)	Translate high-level constructs into lower-level **bytecode** and manage calling conventions.

Test-Suite & Web IDE	Apply **software testing principles** and create reproducible workflows for code execution.

5.4.6 Team Roles and Responsibilities

The successful completion of **Kotha** required expertise in various domains of system programming.

Table 5.9 — Team Roles and Responsibilities

Name	Student ID	Role & Responsibilities
Labony Sur	232-15-473	**Project Lead & Core Architect** • Designed the overall System Architecture and Grammar. • Implemented the **Virtual Machine (VM)** and **Garbage Collector**. • Developed the **IR Generator** and **Optimizer**.
Aupurba Sarker	232-15-269	**Frontend Engineer & Documentation** • Implemented the **Lexer (lexer.l)** and defined Regex patterns. • Managed the **Symbol Table** logic. • Authored the complete **Project Report**.

Alzubair Ahamed	232-15-591	**Tooling & IDE Developer** • Developed the **Web IDE** frontend (HTML/CSS/JS). • Built the Python backend server (server.py). • Created the **Standard Library** functions (math_lib.c).

5.4.7 Collaboration and Tools

To maintain code quality and efficient teamwork, the following tools were utilized:

- **GitHub:** For hosting the repository and managing version control.
  Github Repository link: https://github.com/labonysur-cloud/Kotha0.2
- **VS Code:** The primary editor, utilized with "Live Share" for pair programming.
- **Google Meet:** Used for daily communication and sharing debug logs.
- **Google Drive:** Used for storing diagrams and report drafts.

5.5 Summary

Effective project management was crucial in handling the complexity of building a compiler from scratch. By adhering to a strict schedule, conducting risk analysis, and clearly defining roles, the team successfully delivered a working compiler, VM, and IDE within the allotted timeframe.

Chapter 6

Discussion

This chapter provides a critical analysis of the **Kotha** programming language ecosystem. It evaluates the system's current capabilities against standard industry benchmarks, honestly addresses architectural limitations inherent in the current design, and outlines a concrete, realistic roadmap for future development. By examining both the achievements and the gaps, we establish a clear trajectory for evolving Kotha from an educational prototype into a robust general-purpose language.

6.1 Current Limitations

Based on the current codebase structure, several limitations exist that distinguish Kotha from commercial-grade languages. These are not failures but typical characteristics of a language in its early development stage (v0.2.0).

1. **Limited Data Structures:**
    - **Reality:** The current AST (ast.h) and Symbol Table (symtab.c) support primitive types (int, float, string) and simple integer arrays (talika).
    - **Limitation:** There is no support for complex data structures like **Classes**, **Structs**, **HashMaps**, or **Linked Lists**. This makes it difficult to model real-world entities (e.g., creating a "Student" object with both a name and an ID) without creating multiple parallel arrays.
2. **Basic Error Reporting:**
    - **Reality:** The parser (parser.y) catches syntax errors and stops execution. The VM (vm.c) catches runtime errors like division by zero.
    - **Limitation:** The error messages are generic (e.g., "Syntax Error"). The system lacks a sophisticated "panic mode" recovery or a detailed stack trace that points a user to the exact column number or suggests fixes (e.g., "Did you mean 'dhoro'?").
3. **Single-Threaded Execution:**
    - **Reality:** The Virtual Machine executes bytecode instructions sequentially in a single while loop.
    - **Limitation:** There is no support for concurrency, threading, or asynchronous tasks. Kotha cannot yet utilize modern multi-core processors, making it unsuitable for high-performance parallel computing.
4. **Scope and Modularity:**
    - **Reality:** While the include keyword exists, the compiler largely processes the program as a single translation unit.

- **Limitation:** There is no advanced namespace management. A variable named x in an included file could accidentally overwrite a global variable named x in the main program, leading to potential bugs in larger projects.

6.2 Strategies to Overcome Limitations

We have devised specific technical strategies to address these issues in the next version (v0.3):

- **Implementing Structs:** We will modify the AST to support a new node type NODE_STRUCT_DECL. The Symbol Table will be updated to handle custom types, allowing users to group related data.
- **Enhanced Diagnostics:** We will implement error recovery in Bison. Instead of halting at the first error, the compiler will skip to the next semicolon and continue scanning, allowing it to report multiple errors at once with "Did you mean..." suggestions.
- **Module System:** We will introduce a file-based scope system. When a file is included, its variables will be loaded into a separate "Scope Object" in the compiler, preventing naming collisions.

6.3 Future Plan

The roadmap for Kotha focuses on expanding its utility from an educational tool to a practical language for application development.

6.3.1 Short-Term Goals (v0.3 - v0.5)

1. **Mobile IDE Application:**
    - **Plan:** Port the current Web IDE logic to a mobile application using a framework like **Flutter** or **React Native**.
    - **Goal:** Since the backend is a portable Python script and the compiler is standard C, they can be embedded in Android/iOS environments, allowing students in rural Bangladesh to practice coding directly on smartphones.
2. **Specialized Libraries (KothaML & KothaGame):**
    - **Status:** Keywords for these libraries (like shikha for training, khela for game loops) have already been reserved in the IDE's syntax highlighter (script.js).
    - **Plan:** We will implement the backend logic for these keywords using a **Foreign Function Interface (FFI)**.
        - **KothaML:** Will wrap lightweight C machine learning libraries to allow students to create simple neural networks using Banglish commands.
        - **KothaGame:** Will link to a simple graphics library (like SDL) to allow users to draw shapes and create 2D games using keywords like chitro (image) and sprite.

6.3.2 Long-Term Goals (v1.0+)

1. **Object-Oriented Programming (OOP):**
    - Introducing shreni (class) and bostu (object) keywords to support Encapsulation, Inheritance, and Polymorphism. This is essential for teaching modern software engineering patterns.
2. **Language Server Protocol (LSP):**
    - Currently, Kotha relies on its own custom web IDE. The long-term goal is to build an LSP server. This would allow Kotha to work inside **Visual Studio Code** with full professional features like Autocomplete, "Go to Definition," and Rename Refactoring.

6.6 Conclusion of Discussion

The development of Kotha demonstrates that building a custom, localized compiler is a viable solution to the language barrier in computer science education. While the current version (0.2.0) has limitations regarding data structures and concurrency, the foundational architecture—specifically the separation of the AST, IR, and VM—is robust enough to support these future expansions. By executing this roadmap, Kotha aims to evolve from a "learning toy" into a "learning platform" that empowers the next generation of Bengali engineers.

Chapter 7

Conclusion

7.1 Summary

The **Kotha Programming Language** project was undertaken with a clear vision: to democratize computer science education by breaking down the linguistic barriers that hinder native Bengali speakers. Over the course of this project, we successfully transitioned from a conceptual design to a fully functional, high-performance software ecosystem.

We designed and implemented **Kotha**, a statically-typed, interpreted language that utilizes "Banglish" syntax (e.g., dhoro, jodi, ghuraw). Unlike simple educational tools that function as mere text translators, Kotha is built upon a robust, industry-standard compiler architecture using **C**. The system features a complete execution pipeline:

1. **Lexical Analysis & Parsing:** Utilizing **Flex** and **Bison** to strictly validate syntax and structure.
2. **Intermediate Representation (IR):** Generating machine-independent 3-Address Code to facilitate optimization.
3. **Virtual Machine (VM):** Executing custom bytecode on a stack-based VM featuring a **Mark-and-Sweep Garbage Collector** for automatic memory management.
4. **Integrated Environment:** A fully functional **Web IDE** with a retro interface that allows instant coding without local setup.

7.2 Achievement of Objectives

The project successfully met its primary technical and educational objectives as defined in the introduction:

- **Compiler Construction:** We successfully built a compiler that separates compile-time logic (parsing, IR generation) from runtime logic (VM execution), verifying it is a true compiler architecture and not just an interpreter.
- **Memory Safety:** The implementation of a manual Garbage Collector (vm_gc_collect) ensures efficient memory usage, a complex engineering feat rarely found in undergraduate projects.
- **User Accessibility:** The deployment of the Python-backed Web IDE makes the language immediately accessible to any student with a web browser.

- **Core Functionality:** The language supports all essential programming constructs—variables, arithmetic, control flow (if, while, for), and functions—empowered by a custom C standard library.

7.3 Limitations

While the project is a technical success, a truthful assessment reveals specific constraints inherent to the current version (v0.2.0):

- **Data Structures:** The system currently lacks support for user-defined types such as structs or classes. This limits the ability to model complex real-world data entities.
- **Concurrency:** The Virtual Machine is single-threaded, meaning it cannot perform parallel processing tasks or utilize multi-core processors.
- **Error Diagnostics:** While syntax errors are caught, the error reporting mechanism is basic. It lacks detailed stack traces or "Did you mean?" suggestions, which can make debugging challenging for beginners.
- **Ecosystem:** Compared to established languages, Kotha lacks a package manager and extensive third-party libraries for tasks like networking or advanced graphics.

7.4 Future Work

The modular architecture of Kotha (separating Frontend, Middle-end, and Backend) provides a solid foundation for significant expansion. The immediate roadmap includes:

1. **Mobile App Integration:** Porting the IDE logic to Android and iOS platforms to reach students in rural Bangladesh who rely primarily on smartphones for education.
2. **Professional Tooling:** Developing a Language Server Protocol (LSP) to support Kotha in professional editors like **VS Code**, enabling features like auto-completion and syntax highlighting.
3. **Advanced Libraries:** Implementing a Foreign Function Interface (FFI) to allow Kotha to call external C libraries. This paves the way for the planned **"KothaML"** (Machine Learning) and **"KothaGame"** engines, allowing students to build AI models and games using Banglish syntax.
4. **Object-Oriented Features:** Introducing shreni (class) and bostu (object) keywords to teach modern software design patterns.

7.5 Concluding Remarks

Kotha stands as a testament to the potential of localized technology. It proves that a programming language need not be bound by English syntax to be powerful and efficient. By abstracting complex compiler concepts into a culturally familiar context, Kotha has the potential to accelerate the learning curve for millions of Bengali students. This project is not just a software application; it is a step towards digital inclusivity and a foundation for the next generation of **Bengali** computer scientists.

References

This section lists all books, research papers, documentation, and online resources consulted during the development of the project.

Books and Textbooks

[1] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley. ISBN: 978-0321486813.

[2] Appel, A. W., & Palsberg, J. (2002). *Modern Compiler Implementation in C*. Cambridge University Press. ISBN: 978-0521607650.

[3] Cooper, K. D., & Torczon, L. (2011). *Engineering a Compiler* (2nd ed.). Morgan Kaufmann. ISBN: 978-0120884780.

[4] Levine, J. R. (2009). *flex & bison: Text Processing Tools*. O'Reilly Media. ISBN: 978-0596155971.

[5] Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J., & Langendoen, K. (2012). *Modern Compiler Design* (2nd ed.). Springer. ISBN: 978-1461446989.

[6] Nystrom, R. (2021). *Crafting Interpreters*. Genever Benning. ISBN: 978-0990582939.

Academic Papers and Research Articles

[7] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), 451–490. https://doi.org/10.1145/115372.115320

[8] Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (pp. 75–86). IEEE. https://doi.org/10.1109/CGO.2004.1281665

[9] Shi, Y., Casey, K., Ertl, M. A., & Gregg, D. (2008). Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4(4), 1–36. https://doi.org/10.1145/1328195.1328197

[10] Brunthaler, S. (2010). Inline caching meets quickening. In *ECOOP 2010 – Object-Oriented Programming* (pp. 429–451). Springer. https://doi.org/10.1007/978-3-642-14107-2_21

Technical Documentation and Standards

[11] GNU Project. (2023). *Bison – GNU parser generator*. Free Software Foundation. Retrieved from https://www.gnu.org/software/bison/

[12] Paxson, V., & Estes, W. (2023). *Flex: The Fast Lexical Analyzer*. Retrieved from https://github.com/westes/flex

[13] Free Software Foundation. (2023). *GCC, the GNU Compiler Collection*. Retrieved from https://gcc.gnu.org/

[14] ISO/IEC. (2018). *ISO/IEC 9899:2018 – Programming Languages — C*. International Organization for Standardization.

Web Development and IDE Technologies

[15] Mozilla Developer Network. (2023). *JavaScript Guide*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

[16] Python Software Foundation. (2023). *Python 3 Documentation*. Retrieved from https://docs.python.org/3/

[17] W3C. (2023). *HTML Living Standard*. Retrieved from https://html.spec.whatwg.org/

[18] W3C. (2023). *CSS Specifications*. Retrieved from https://www.w3.org/Style/CSS/

Programming Language Design

[19] Ierusalimschy, R., De Figueiredo, L. H., & Celes, W. (2007). The evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (pp. 2-1–2-26). ACM. https://doi.org/10.1145/1238844.1238846

[20] Odersky, M., Spoon, L., & Venners, B. (2008). *Programming in Scala*. Artima Press. ISBN: 978-0981531601.

[21] Sebesta, R. W. (2015). *Concepts of Programming Languages* (11th ed.). Pearson. ISBN: 978-0133943023.

Virtual Machines and Runtime Systems

[22] Lindholm, T., Yellin, F., Bracha, G., & Buckley, A. (2014). *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley. ISBN: 978-0133260441.

[23] Smith, J. E., & Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann. ISBN: 978-1558609105.

[24] Ertl, M. A., & Gregg, D. (2003). The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5, 1–25.

Optimization Techniques

[25] Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann. ISBN: 978-1558603202.

[26] Allen, F. E. (1970). Control flow analysis. *ACM SIGPLAN Notices*, 5(7), 1–19. https://doi.org/10.1145/390013.808479

[27] Knoop, J., Rüthing, O., & Steffen, B. (1994). Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4), 1117–1155. https://doi.org/10.1145/183432.183443

Symbol Tables and Type Systems

[28] Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press. ISBN: 978-0262162098.

[29] Cardelli, L., & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), 471–523. https://doi.org/10.1145/6041.6042

Localization and Internationalization

[30] Unicode Consortium. (2023). *The Unicode Standard, Version 15.0*. Retrieved from https://www.unicode.org/versions/Unicode15.0.0/

[31] Ishida, R. (2023). *Internationalization Techniques: Authoring HTML & CSS*. W3C. Retrieved from https://www.w3.org/International/techniques/authoring-html

Software Engineering and Development Tools

[32] Chacon, S., & Straub, B. (2014). *Pro Git* (2nd ed.). Apress. ISBN: 978-1484200773.

[33] Stallman, R. M., McGrath, R., & Smith, P. D. (2004). *GNU Make: A Program for Directing Recompilation*. Free Software Foundation.

[34] Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. ISBN: 978-0201616224.

REPL and Interactive Programming

[35] McDirmid, S. (2013). Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (pp. 53–62). ACM. https://doi.org/10.1145/2509578.2509585

[36] Tanimoto, S. L. (2013). A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming* (pp. 31–34). IEEE. https://doi.org/10.1109/LIVE.2013.6617346

Parsing and Lexical Analysis

[37] Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2), 94–102. https://doi.org/10.1145/362007.362035

[38] Aho, A. V., & Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall. ISBN: 978-0139145568.

[39] Grune, D., & Jacobs, C. J. (2008). *Parsing Techniques: A Practical Guide* (2nd ed.). Springer. ISBN: 978-0387202488.

Code Generation and Intermediate Representations

[40] Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1988). Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 12–27). ACM. https://doi.org/10.1145/73560.73562

[41] Click, C., & Cooper, K. D. (1995). Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2), 181–196. https://doi.org/10.1145/201059.201061

Memory Management

[42] Jones, R., Hosking, A., & Moss, E. (2011). *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Press. ISBN: 978-1420082791.

[43] Wilson, P. R. (1992). Uniprocessor garbage collection techniques. In *International Workshop on Memory Management* (pp. 1–42). Springer. https://doi.org/10.1007/BFb0017182

Online Resources and Tutorials

[44] Nystrom, R. (2023). *Crafting Interpreters Online*. Retrieved from https://craftinginterpreters.com/

[45] Stack Overflow. (2023). *Compiler Construction*. Retrieved from https://stackoverflow.com/questions/tagged/compiler-construction

[46] GeeksforGeeks. (2023). *Compiler Design Tutorials*. Retrieved from https://www.geeksforgeeks.org/compiler-design-tutorials/

[47] Tutorialspoint. (2023). *Compiler Design Tutorial*. Retrieved from https://www.tutorialspoint.com/compiler_design/

Bengali Language and Computing

[48] Chowdhury, S. A., & Alam, M. J. (2014). Bangla text processing: A comprehensive review. *International Journal of Computer Applications*, 97(20), 1–8. https://doi.org/10.5120/17114-7534

[49] Das, A., & Bandyopadhyay, S. (2010). Morphological stemming cluster identification for Bangla. In *Proceedings of the 1st Workshop on South and Southeast Asian Natural Language Processing* (pp. 1–8). ACL.

[50] Rahman, M., & Bhuiyan, M. (2012). Challenges and opportunities in Bengali language processing. *Journal of Computing*, 4(2), 45–52.