AY 2024-25

IV Semester

CSL406

PROJECT REPORT
On

BadeBhai-Lang

Bachelor of Technology
in
Computer Science and Engineering

By
Ujjwal Mishra
[23162]

SCHOOL OF COMPUTING

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY UNA
HIMACHAL PRADESH

MAY 2025

BONAFIDE CERTIFICATE

This is to certify that the project titled *BadeBhai-Lang* is a bonafide record of the work done by

Ujjwal Mishra [23162]

in partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering of the INDIAN INSTITUTE OF INFORMATION TECHNOLOGY UNA, HIMACHAL PRADESH, during the year 2024 - 2025.

under the guidance of

Dr. Ashish Maurya

Project viva-voce held on: _____

Internal Examiner External Examiner

ORIGNALITY / NO PLAGARISM DECLARATION

I certify that this project report is our original report and no part of it is copied from any published reports, papers, books, articles, etc. I certify that all the contents in this report are based on our personal findings and research and we have cited all the relevant sources which have been required in the preparation of this project report, whether they be books, articles, reports, lecture notes, and any other kind of document. I also certify that this report has not previously been submitted partially or as whole for the award of degree in any other university in India and/or abroad.

I hereby declare that, I am fully aware of what constitutes plagiarism and understand that if it is found at a later stage to contain any instance of plagiarism, my degree may be cancelled.

Ujjwal Mishra [23162]

ABSTRACT

BadeBhai-lang is a dynamically typed, object-oriented programming language designed specifically to make programming more accessible for non-native English speakers, particularly Hindi speakers, by utilizing Romanized Hindi syntax. Built using Rust, BadeBhai-lang follows the structure and implementation strategies outlined in Robert Nystrom's *Crafting Interpreters*, implementing a tree-walk interpreter architecture.

The core motivation behind BadeBhai-lang is to lower the entry barrier to programming by simplifying syntax and using familiar linguistic constructs. By replacing conventional English-based keywords with Romanized Hindi equivalents—such as yehai for variable declaration, dikaho for print statements, and jabtak for loops—the language aims to create a comfortable and intuitive learning curve for beginners. BadeBhai-lang supports key programming paradigms including imperative programming, procedural abstraction through functions, and object-oriented design through classes and inheritance.

Internally, the language architecture is structured into scanning, parsing, static analysis, and interpretation phases. Dynamic typing is employed to enhance flexibility, and memory management is handled using a mark-and-sweep garbage collector, ensuring robust runtime behavior without manual intervention. Rust's guarantees of memory safety, zero-cost abstractions, and concurrency support make it an ideal choice for implementing the interpreter backend.

BadeBhai-lang successfully demonstrates how localization of programming languages can play a vital role in making computational thinking more accessible. The project not only showcases language design and interpreter implementation skills but also addresses a larger educational goal of broadening programming literacy among linguistically diverse populations. Future expansions include optimization through bytecode compilation, module systems, extended standard libraries, and the development of supporting tooling such as syntax-aware editors.

Through this project, BadeBhai-lang represents a pioneering step towards creating culturally and linguistically inclusive programming environments in India.

***Keywords*:** Object-Oriented Programming; Programming Language Theory; Indian Language Programming; Tree-walk Interpreter.

ACKNOWLEDGEMENT

I would like to thank the following people for their support and guidance without whom the completion of this project in fruition would not be possible.

I would like to express our sincere gratitude and heartfelt thanks to Dr. Ashish Maurya for their unflinching support and guidance, valuable suggestions and expert advice. His words of wisdom and expertise in subject matter were of immense help throughout the duration of this project.

I also take the opportunity to thank our Director and all the faculty of School of Computing, IIIT Una for helping me by providing necessary knowledge base and resources.

I would also like to thank my parents and friends for their constant support.

Ujjwal Mishra [23162]

TABLE OF CONTENTS

Title **Page No.**

ABSTRACT……iii

ACKNOWLEGEMENT……iv

TABLE OF CONTENTS……v

LIST OF ACRONYMS……..vi

LIST OF FIGURES……..vii

1 Introduction……...1

2 Review of Literature ……2

2.1 Introduction……...……..2

2.2 Historical Background…...…….....……...……..2

2.3 Theoretical Foundations..…….....……...……..3

2.4 Current Work and Innovations.....……..4

3 Motivation and Problem Definition……6

4 Proposed Methodology ……7

4.1 Vocabulary and Syntax Overview.....……..7

4.2 Syntax Design Philosophy.....……..8

4.3 Data Types.....……..8

4.4 Expressions and Operators.....……...9

4.5 Statements and Control Flow.....…….……..9

4.6 Functions.....…….……10

4.7 Classes and Object-Oriented Features.....…….……..10

4.8 Memory Management and Garbage Collection.....…….……11

4.9 Interpreter Architecture.....…….……..11

4.10 Implementation Details.....…….……..12

4.11 Advantages of Using Rust.....…….……..12

5 Conclusion and Future Work……13

References……...……14

Appendices…….15

LIST OF ACRONYMS

AST	Abstract Syntax Tree
CFG	Context Free Grammer
FFI	Foreign Function Interface
JIT	Just In Time
OOP	Object Oriented Programming
SQL	Structured Query Language

LIST OF FIGURES

2.1 A 'Path' to language building as depicted in Crafting Interpreters.…….....6

4.1 A visual representation of the AST for 1+2.....……12

Chapter 1

Introduction

Programming has long been perceived as an intimidating domain, particularly for individuals who are not fluent in English or who lack prior exposure to the highly structured and abstract syntax of conventional programming languages. Despite the massive growth of the Indian technology industry, much of programming education and practice remains inaccessible to a significant portion of the population, largely due to linguistic and cultural barriers. Recognizing this gap, **BadeBhai-lang** was conceived: a novel programming language designed specifically to make programming concepts approachable, relatable, and easy to grasp for Hindi-speaking learners.

BadeBhai-lang is a **tree-walk interpreted programming language**, meticulously built in **Rust**, one of the most modern and safe systems programming languages. It draws heavily from the theoretical foundations presented in *Crafting Interpreters* by Robert Nystrom[1] while adapting and enhancing them to leverage Rust's[3] powerful memory safety guarantees and concurrency features. The language incorporates support for **object-oriented programming** paradigms, **automatic garbage collection**, and **dynamic typing**, offering users a rich, yet manageable, learning environment.

The distinguishing feature of BadeBhai-lang is its **romanized Hindi syntax**. Rather than using traditional English-based keywords and programming constructs, BadeBhai-lang introduces commands and control structures in colloquial Hindi, written in the Roman alphabet. For instance, instead of `print`, the language uses *`dikhao`*, and instead of `if`/`else`, expressions like *`agar`*/*`varna`* could be utilized. This shift not only bridges the linguistic gap but also lowers the cognitive load required to understand core programming concepts, making coding feel more natural and less alien to first-time learners.

From an architectural standpoint, BadeBhai-lang follows a classical **interpreter pipeline**: it first tokenizes the input source code (scanning phase), builds an **Abstract Syntax Tree (AST)** from the tokens (parsing phase), performs **semantic analysis** such as scope resolution and type checks, and finally **interprets** the AST by traversing it node-by-node. Despite using a **tree-walk interpretation** model — which is generally slower than bytecode virtual machines — the focus remains on educational clarity and development simplicity rather than execution speed.

Choosing Rust as the implementation language posed unique challenges, particularly in managing variable lifetimes and shared ownership across complex structures like scopes, classes, and functions. Nonetheless, Rust's robust safety features ultimately contributed to a highly reliable and performant interpreter core.

Chapter 2

Review of Literature

2.1 Introduction

Programming language development has historically been a complex task reserved for advanced practitioners due to the intricate design, compilation, and execution processes involved. However, efforts such as *Crafting Interpreters* by Robert Nystrom[1] have significantly lowered the barriers for independent development of interpreters and compilers. Inspired by this movement, BadeBhai-lang aims to bridge another important gap: making programming more accessible to Hindi-speaking, non-technical audiences by combining simple romanized Hindi syntax with modern language design principles.

2.2 Historical Background

The history of programming languages is rooted in an ongoing effort to simplify human-computer interaction. Early languages like **FORTRAN** (1957) and **COBOL** (1959) marked the first major step away from machine code towards more human-readable syntax. Over the decades, languages evolved: procedural languages like **C** (1972), object-oriented languages like **C++** (1985) and **Java** (1995), and dynamic, interpreted scripting languages like **Python** (1991) and **JavaScript** (1995).

An important distinction emerged in how languages were executed: whether they were **compiled** or **interpreted**.

- **Compiled Languages**: These are languages where the source code is translated into machine code ahead of time by a compiler. The resulting executable is run directly by the operating system. Examples include **C**, **C++**, and **Rust**. Compiled languages are generally faster at runtime because the translation step has already been performed.
- **Interpreted Languages**: Here, the source code is executed line-by-line or statement-by-statement by an interpreter without producing a separate executable. Languages like **Python**, **Ruby**, and **JavaScript** are interpreted. This makes them easier to develop and debug but often slower in execution compared to compiled languages.
- **Tree-Walk Interpreters**: A specific kind of interpreter, tree-walk interpreters, first parse the entire program into an **Abstract Syntax Tree (AST)** and then recursively "walk" this tree to execute code. Tree-walk interpreters are simple to implement and great for learning or prototyping new languages. However, they are slower compared

to bytecode virtual machines or Just-In-Time (JIT) compiled runtimes because every expression involves runtime traversal of complex tree structures.

Understanding these execution models is crucial when building a new language. BadeBhai-lang, designed as a **tree-walk interpreted language**, prioritizes ease of implementation and educational simplicity over raw performance — an appropriate trade-off for its intended audience of beginners and non-programmers.

Meanwhile, throughout the history of programming education, various initiatives like **Logo** (1967), **BASIC** (1964), and later **Scratch** (2003) have tried to make programming accessible to new learners. However, these efforts remained largely English-centric. BadeBhai-lang uniquely addresses a linguistic gap, providing the first attempt at a romanized Hindi syntax programming language, being a successor to **BhaiLang**, another Tree-Walk Interpreted langugae with romanized Hindi syntax that was developed in 2020.

2.3 Theoretical Foundations

The foundation of BadeBhai-lang's design rests on a combination of classic compiler theory and modern programming practices, adapted to the requirements of accessibility and beginner-friendliness.

At its heart, the project follows the standard phases of a programming language interpreter pipeline:

- **Scanning (Lexical Analysis)**: The raw source code is first scanned into a stream of tokens, removing comments and whitespace, and identifying meaningful symbols like keywords, operators, identifiers, and literals.
- **Parsing (Syntactic Analysis)**: The token stream is then parsed according to a predefined grammar into an **Abstract Syntax Tree (AST)**. The AST represents the hierarchical structure of the program, with nodes corresponding to language constructs like expressions, statements, and function definitions.
- **Static Analysis**: In this optional but powerful step, the AST is traversed to perform checks like scope resolution, type checking (if applicable), and other semantic validations.
- **Interpretation**: Finally, the AST is traversed recursively in a **tree-walk interpreter** manner. Each node is evaluated according to its semantics, executing the program dynamically.

These stages are outlined extensively in *Crafting Interpreters*[1]and supplemented by concepts from *Compilers: Principles, Techniques, and Tools* ("Dragon Book")[2] as depicted in fig 1.

2.3.1 Tree-Walk Interpretation and Its Implications

Tree-walk interpreters offer simplicity, readability, and easier debugging — making them ideal for educational and small language projects. However, they are less performant for computationally heavy programs. The overhead of recursive calls and AST traversal means that a tree-walk interpreter is typically 10–100x slower than a compiled language. For BadeBhai-lang's intended audience, this is an acceptable trade-off: clarity and simplicity over raw speed.

2.3.2 Rust-Specific Adaptation

Implementing a tree-walk interpreter in Rust[3] posed unique challenges. Rust enforces strict rules about ownership, borrowing, and lifetimes to guarantee memory safety. Most language runtimes (especially garbage-collected ones) require flexible, cyclic data structures, which clash with Rust's default linear ownership model.

2.3.3 Programming Language Theory Aspects

BadeBhai-lang also embeds principles from formal language theory:

- **Context-Free Grammars (CFGs)** define the syntactic structure.
- **Recursive Descent Parsing** strategies are used for building the AST.
- **Lexical scoping** principles govern variable resolution and shadowing behavior.
- **Dynamically typed runtime semantics** handle type safety checks during execution rather than at compile time.

2.4 Current Work and Innovations

Modern language projects often fall into two categories: domain-specific scripting languages (e.g., Lua, SQL) or general-purpose systems languages (e.g., Go, Rust). Efforts like **Pinecone**, **Wren**, and **BhaiLang** (all small, personal language projects) have gained popularity in educational circles.

BadeBhai-lang situates itself uniquely among these efforts:

- **Language Syntax Localization**: Using romanized Hindi makes the language much more approachable for non-English-speaking users — an innovation scarcely explored elsewhere.
- **Rust Implementation**: Unlike Lox from *Crafting Interpreters* (implemented in Java and C), BadeBhai-lang is entirely written in Rust, providing benefits such as thread safety, memory safety, and performance optimization.
- **Object-Oriented Structure with Simple Semantics**: While many beginner-friendly languages (like Python) are object-oriented, BadeBhai-lang simplifies object usage further by using familiar Hindi metaphors in naming and syntax.

In contrast with existing educational languages, BadeBhai-lang emphasizes not just ease of syntax but cultural familiarity.

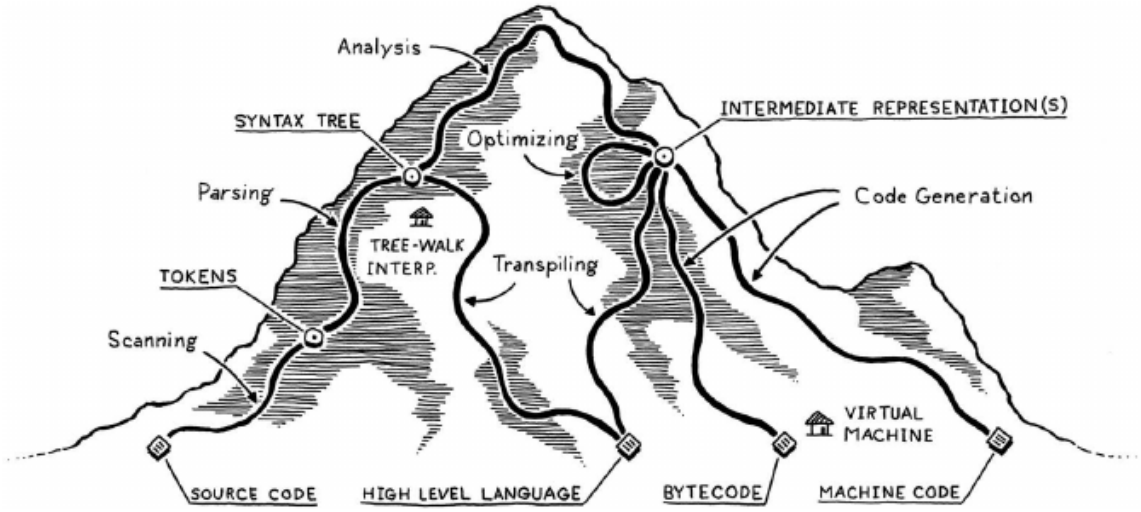


Figure 2.1: A 'Path' to language building as depicted in Crafting Interpreters.

Chapter 3

Motivation and Problem Defination

In India, the medium of primary education, particularly in government and regional schools, is predominantly in local languages such as Hindi, Tamil, Bengali, Marathi, and others. English is typically introduced as a secondary language in later classes, often starting from Grade 4 or even later depending on the region. Consequently, for a vast majority of students, fluency and comfort with English — especially technical English — comes much later in their educational journey, if at all.
When these students are introduced to programming languages like C, C++, Java, or Python, they encounter not just the challenge of learning new computational concepts, but also the parallel burden of decoding unfamiliar English-based syntax. Keywords such as `if`, `else`, `while`, `return`, and `function` — though seemingly simple for fluent English speakers — act as cognitive hurdles for those who are still developing English proficiency. Instead of focusing on essential skills like logical reasoning, algorithmic thinking, and problem decomposition, students expend a disproportionate amount of effort on memorizing rigid syntax patterns and fighting language barriers.

This creates an unfortunate scenario where students equate programming with rote memorization rather than creative problem-solving. They become anxious about small syntax mistakes like missing semicolons, parentheses, or misspelled keywords, and this anxiety often overshadows their ability to think systematically and design logical solutions. As a result, many lose confidence early and develop a long-lasting fear of coding, which could otherwise have been a gateway to opportunities in the modern digital economy.

BadeBhai-lang is designed to address this deeply rooted issue. By using **romanized Hindi syntax**, it removes the psychological barrier imposed by foreign vocabulary and allows students to concentrate on what truly matters — building logic and understanding how computers "think." By mapping familiar Hindi phrases to programming constructs, the language offers a more intuitive entry point into computational thinking.

The motivation behind BadeBhai-lang is not just to teach programming, but to build **confidence** and **logical clarity** in students from diverse linguistic backgrounds. It strives to make the first encounter with programming an empowering experience, helping students transition smoothly to other languages later with a strong foundation in logic, rather than frustration with syntax.

Chapter 4

Proposed Methodology

4.1 Vocabulary and Syntax Overview

BadeBhai-lang is a dynamically typed, tree-walk interpreted language, designed to reduce the barrier to programming for Hindi-speaking audiences. Inspired by the structure explained in "Crafting Interpreters", BadeBhai-lang covers all essential elements of a full-featured programming language. It borrows syntax familiarity from C-family languages but adapts keywords to Romanized Hindi for accessibility.

4.1.1 Keywords Mapping:

- aur → Logical AND
- ya → Logical OR
- class → Class Declaration
- else → Alternative branch
- galat → Boolean false
- sahi → Boolean true
- for → For loop
- fun → Function declaration
- if → Conditional branching
- nil → Null value
- dikaho → Print statement
- return → Return from function
- super → Superclass method access
- this → Current object instance
- yehai → Variable declaration
- jabtak → While loop
- ruko → Break from loop

Identifiers consist of alphabetic characters, digits, and underscores but must not start with a digit. Comments begin with // and extend to the end of the line. Reserved keywords cannot be

used as variable names. String interpolation, inline comments, and doc-comments are potential future features to enhance expressiveness.

4.2 Syntax Design Philosophy

The choice of Romanized Hindi and natural keywords enhances the intuitiveness of the syntax. By minimizing unfamiliar symbols and adopting keyword-based logical operators, the language lowers cognitive load for beginners. Syntax consistency across constructs such as functions, classes, and control flows ensures predictability, improving learnability.

4.3 Data Types

Programming languages use data types as building blocks to represent information. BadeBhai-lang provides a minimal yet sufficient set of primitive types to build complex programs.

4.3.1 Booleans

- Represented by sahi and galat.
- Logical operations conform to short-circuit strategies: aur (AND) and ya (OR).
- Booleans are essential for control flow mechanisms and logical validations.
- Truthiness: all values except galat and nil are considered sahi in conditional expressions.

4.3.2 Numbers

- Only floating-point numbers (double-precision, IEEE 754) are supported.
- Integer literals like 10 and floating literals like 10.5 are valid.
- Implicit type promotions simplify user experience.
- Future enhancements could include explicit integer and decimal types.

4.3.3 Strings

- Strings are sequences of Unicode characters, allowing multi-language text.
- Escape sequences like \n, \t, \", and \\ are recognized.
- String operations are crucial for user interaction, formatted output, and data manipulation.

4.3.4 Nil

- nil signifies absence of a value.
- Prevents undefined behavior by providing a default for uninitialized variables.

- Acts as a bottom value in type systems, easing error handling.

4.4 Expressions and Operators

Expressions form the computational core of BadeBhai-lang. Each expression reduces to a value, which can be used or discarded.

4.4.1 Arithmetic Operators

- Fundamental operations: Addition (+), Subtraction (-), Multiplication (*), Division (/).
- Operator precedence follows standard algebraic rules.
- Errors like division by zero are runtime-checked.

4.4.2 Comparison Operators

- Relational comparisons include <, <=, >, >=.
- Equality checks are strict: values must have the same type and value to be considered equal.
- No implicit type coercions during comparisons avoid subtle bugs.

4.4.3 Logical Operators

- aur and ya are lazy evaluated: right-hand side evaluated only if necessary.
- Logical NOT ! negates a Boolean value.
- Logical expressions control conditional branching and loops.

4.4.4 Grouping

- Parentheses enforce explicit precedence in complex expressions.
- Nested parentheses are supported to any depth.

4.5 Statements and Control Flow

Statements form the backbone of the program's execution. They manage evaluation order and side effects.

4.5.1 Variable Declaration

- Variables are introduced using yehai.
- Scope rules ensure lexical scoping; variables exist only within the block they are declared.

4.5.2 Print Statement

- dikaho evaluates its argument and displays the result.
- Useful for debugging and user interaction.
- Non-string values are automatically stringified.

4.5.3 If Statement

- If-else constructs model branching logic.
- Nested if-else structures enable complex decision trees.

4.5.4 While Loop

- Jabtak loops model repetitive execution based on a dynamic condition.
- Breaks out using ruko.

4.5.5 For Loop

- A syntactic sugar built upon while loops.
- Introduces a concise form for iteration with initializer, condition, and increment parts.

4.5.6 Break Statement

- Used within loops to terminate iteration prematurely.
- Enhances flexibility for early exits based on runtime conditions.

4.6 Functions

Functions are essential abstractions that enable modular and reusable code.

4.6.1 Function Declaration

- Functions are declared using fun keyword.
- Can capture variables from the enclosing scope forming closures.
- Support for higher-order functions by treating functions as first-class citizens.

4.6.2 Return Statement

- Functions can return a value using return.
- Omitting return defaults to nil.

4.7 Classes and Object-Oriented Features

Object-oriented programming (OOP) allows modeling real-world entities and behaviors.

4.7.1 Class Declaration

- Classes aggregate methods.
- Instances (objects) are dynamically created and manipulated.

4.7.2 this Keyword

- References the receiver object within methods.
- Helps differentiate between local variables and instance fields.

4.7.3 Inheritance

- Classes can inherit from a single superclass.
- Superclass methods are invoked using super.
- Encourages code reuse and polymorphism.

4.8 Memory Management and Garbage Collection

Memory management is critical for stability and performance.

- A mark-and-sweep garbage collector is used.
- Dead objects are reclaimed automatically without programmer intervention.
- Objects become unreachable when no active references exist, marking them for cleanup.

4.9 Interpreter Architecture

BadeBhai-lang interpreter adopts a modular and layered architecture to maintain clarity and scalability.

4.9.1 Scanning Phase

- Scans raw source text and produces a stream of tokens.
- Recognizes keywords, identifiers, numbers, strings, and symbols.

4.9.2 Parsing Phase

- Converts token streams into an Abstract Syntax Tree (AST).
- Enforces grammatical correctness and operator precedence.

4.9.3 Static Analysis Phase

- Resolves variable and function scopes.
- Identifies semantic errors such as undefined variables or duplicate declarations.

4.9.4 Interpretation Phase

- Traverses AST nodes to evaluate expressions and execute statements.
- Manages variable environments, runtime stacks, and dynamic dispatch.

4.10 Implementation Details

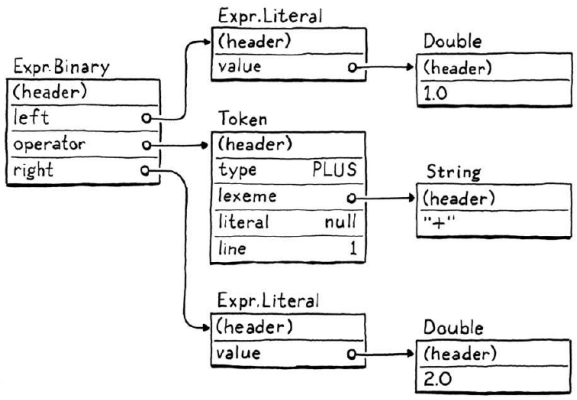
The interpreter is built in Rust, a system programming language offering performance and safety.

- Code modularized into scanner, parser, resolver, interpreter, and runtime components.
- Robust error reporting mechanisms integrated at each phase.
- Lexical environments implemented as chained hashmaps.
- Classes and instances modeled using dynamic trait objects.

4.11 Advantages of Using Rust

Rust offers multiple advantages over traditional implementation languages:

- **Memory Safety**: No null pointers, buffer overflows, or dangling references.
- **Zero-cost Abstractions**: High-level constructs do not impact performance.
- **Fearless Concurrency**: Race conditions are caught at compile time.
- **Pattern Matching**: Elegant handling of variant types and complex data structures.
- **Superior Tooling**: Cargo package manager and integrated formatting, linting tools.
- **Ecosystem**: Rich libraries for everything from parsing (nom) to web development (actix).



Flow Diagram 4.1: *A visual representation of the AST for 1+2*

Chapter 5

Conclusion and Future Work

5.1 Future work

- **Optimization**: Develop a bytecode virtual machine to increase execution speed.
- **Standard Library Expansion**: Introduce built-in support for collections, file I/O, and networking.
- **Concurrency Support**: Add coroutines and async/await constructs.
- **Improved Tooling**: Develop a dedicated IDE with syntax highlighting and auto-completion.
- **Advanced Error Handling**: Better diagnostics, backtrace reporting, and panic recovery.
- **Interoperability**: Build a foreign function interface (FFI) to call C and Rust libraries.
- **Community Building**: Open-source the project and encourage community contributions.
- **Formal Specification**: Publish a complete language specification to standardize future implementations.
- **Educational Material**: Develop tutorials, books, and online courses tailored to Indian students.

5.1 Conclusion

BadeBhai-lang successfully implements a powerful yet approachable programming language using Rust. It captures all major aspects of language design, including a clean syntax, dynamic typing, garbage collection, and object-oriented programming. Built using principles detailed in "Crafting Interpreters," BadeBhai-lang extends its goals by making programming more culturally and linguistically inclusive.

References

[1] Nystrom, R. (2021). Crafting Interpreters. Genever Benning.

[2] Aho, A., Lam, M., Sethi, R., & Ullman, J. (2006). Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc..

[3] The Rust Book- Rust documentation

[4] Hopcroft, J. E., R. Motwani, and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Boston MA, 2006.

[5] Bergin, T. J. and R. G. Gibson, History of Programming Languages, ACM Press, New ` York, 1996

[6] Sethi, R., Programming Languages: Concepts and Constructs, Addison- Wesley, 1996.

[7] McClure, R. M., \TMG | a syntax-directed compiler," Proc. 20th ACM Natl. Conf. (1965), pp. 262{274}.

[8] Chomsky, N., \Three models for the description of language," IRE Trans. on Information Theory IT-2:3 (1956), pp. 113{124.}

[9] Irons, E. T., \A syntax directed compiler for Algol 60," Comm. ACM 4:1(1961), pp. 51{55.}

[10] Earley, J., \Ambiguity and precedence in syntax description," Acta Informatica 4:2 (1975), pp. 183{192}.

[11] Irons, E. T., \A syntax directed compiler for Algol 60," Comm. ACM 4:1 (1961), pp. 51{55.}

[12] Ritchie, D. M., \A tour through the UNIX C compiler," Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979

Appendices

Appendix A

Code Attachments

The following are some examples of codes in BadeBhai-Lang.

A.1 Sample Code to write “hello world”

```
dikhao(“hello world”)
```

OUTPUT: Hello world

A.2 Sample Code to Check Even or Odd

```
fun evenYaOdd(n) {
    if (n % 2 == 0) {
        dikaho "Sankhya Sam (Even) hai.";
    } else {
        dikaho "Sankhya Visham (Odd) hai.";
    }
}

yehai number = 7;
evenYaOdd(number);
```

OUTPUT: Sankhya Visham (Odd) hai

A.3 Sample Code for Simple Class and Object Usage

```
class Aadmi {
    fun parichay() {
        dikaho "Mera naam BadeBhai hai!";
    }
}

yehai aadmi = Aadmi();
aadmi.parichay();
```

OUTPUT: Mera naam BadeBhai hai!