Islamic University of Technology

Department of Computer Science and Engineering (CSE)

B.Sc. in Software Engineering

SWE 4304: Software Project Lab I

Winter, 2024-2025

Project Report

The 'Drim' Programming Language

Group 7

Md. Muntahi Hasan Akhiar, 230042118

S.M. Tahsinuzzaman Emon, 230042104

Tamim Mahrush Naser, 230042133

Supervisor

Farzana Tabassum,

Lecturer, CSE

Date: December 13, 2025

Contents

The ‘Drim’ Programming Language

1 Project Overview

“Drim” is a custom-built, interpreted programming language designed to run as a lightweight, headless command-line interface (CLI) application. Unlike standard compilers that translate source code into machine code, Drim functions as a **Tree-Walk Interpreter** that reads custom `.drim` source files, constructs an internal Abstract Syntax Tree (AST), and executes logic in real-time.

The project is built entirely from scratch using **C++ (Standard 17)**, adhering to strict constraints that prohibit the use of external parsing libraries (like Lex or Yacc) or heavy standard library functions (like `std::stoi` or Regex). This ensures that every component of the language pipeline—from lexical analysis to memory management—is engineered manually by the team.

The primary goal of the project is to create a functional programming environment that supports dynamic typing, mathematical operations, and custom control flow structures (such as `drimming` for loops and `wake` for output), providing a simplified yet robust platform for logic execution.

2 Motivation Behind the Project

The motivation behind developing the Drim programming language is twofold: preserving classroom culture and mastering complex software engineering concepts.

Cultural Significance

The name ”Drim” originated as an inside joke and a shared meme within the student batch. Rather than letting this remain a fleeting concept, we decided to immortalize it by building a tangible software tool. This decision transforms the abstract nature of coding assignments into a personalized project where the syntax and keywords (such as drimming for loops) reflect the team's unique personality. This connection increases developer engagement and passion for the project.

Engineering Challenges

From a technical perspective, this project serves as a rigorous exercise in system-level programming. We chose to build an interpreter from scratch to accomplish three specific learning outcomes:

- **Deep Understanding of Language Design:** By manually implementing lexical analysis, parsing, and interpretation, we gain firsthand experience with the inner workings of programming languages.
- **Memory Management Proficiency:** The project requires careful handling of dynamic memory allocation and deallocation, fostering a strong grasp of pointers and resource management in C++.
- **Problem-Solving Skills:** Overcoming the constraints of not using third-party libraries or certain STL functions challenges us to devise innovative solutions for common programming tasks.
- **Collaboration and Project Management:** Working as a group on a complex software project enhances teamwork, version control practices, and project planning skills.

3 User Requirement

Functional Requirements

- **Input/Output:** The system must accept user input via the `drim()` command and display formatted output using the `wake()` command.
- **Variable Management:** Users must be able to declare variables, assign values, and update them dynamically. The system must support implicit type casting between Strings and Integers.
- **Mathematical Operations:** The interpreter must correctly evaluate arithmetic expressions (`+`, `-`, `*`, `/`, `%`) respecting standard operator precedence.
- **Control Flow:** The language must support conditional logic (`if/else`) and iterative loops (`drimming`) to execute repetitive tasks.
- **Script Execution:** The application must be able to read a text file with the `.drim` extension and execute the code contained within.

Non-Functional Requirements

- **Headless Application:** The system must run entirely in the terminal (CLI) without a graphical user interface.
- **Performance:** The interpreter should parse and execute commands with minimal latency for standard script sizes.
- **Portability:** The source code should be cross-platform, compilable on Windows, macOS, and Linux using CMake.
- **Independence:** The implementation must not rely on any third-party parsing libraries or forbidden STL functions.

4 Key Feature

The Drim language includes a set of custom features designed to balance simplicity with robust logic execution. These features highlight the capabilities of the custom-built Tree-Walk Interpreter.

- **Custom Syntax & Semantics:** The language utilizes a unique set of keywords tailored to the project's theme. Notable examples include the `drimming` command for iterative loops and the `wake()` function for standard output. This custom syntax demonstrates the flexibility of the Lexer design.
- **String Interpolation:** Unlike many basic custom languages that only supports simple stgring printing, Drim supports advanced intercpolation. Users can embed variables directly within string literals (e.g., `wake("Value: x))`, allowing for dyunamic and readable output generation.
- **Dynamic Typing:** Variables in **Drim** can hold either String or Integer values, with implicit type casting handled by the interpreter. This feature simplifies variable management and enhances user experience.
- **Headless Execution:** Designed specifically as a backend tool, Drim runs as a lightweight, headless Command Line Interface (CLI) application. It executes scripts directly in the terminal without the overhead of a Graphical User Interface (GUI), meeting the requirement for high-performance logic processing.
- **Zero-Dependency Architecture:** A defining feature of Drim is its independence. It does not rely on third-party parsing libraries or heavy standard library tools (such as Regex). Every component, from tokenization to the Abstract Syntax Tree (AST) construction, is engineered manually using C++ 17.

5 Flow Chart/Class Diagram

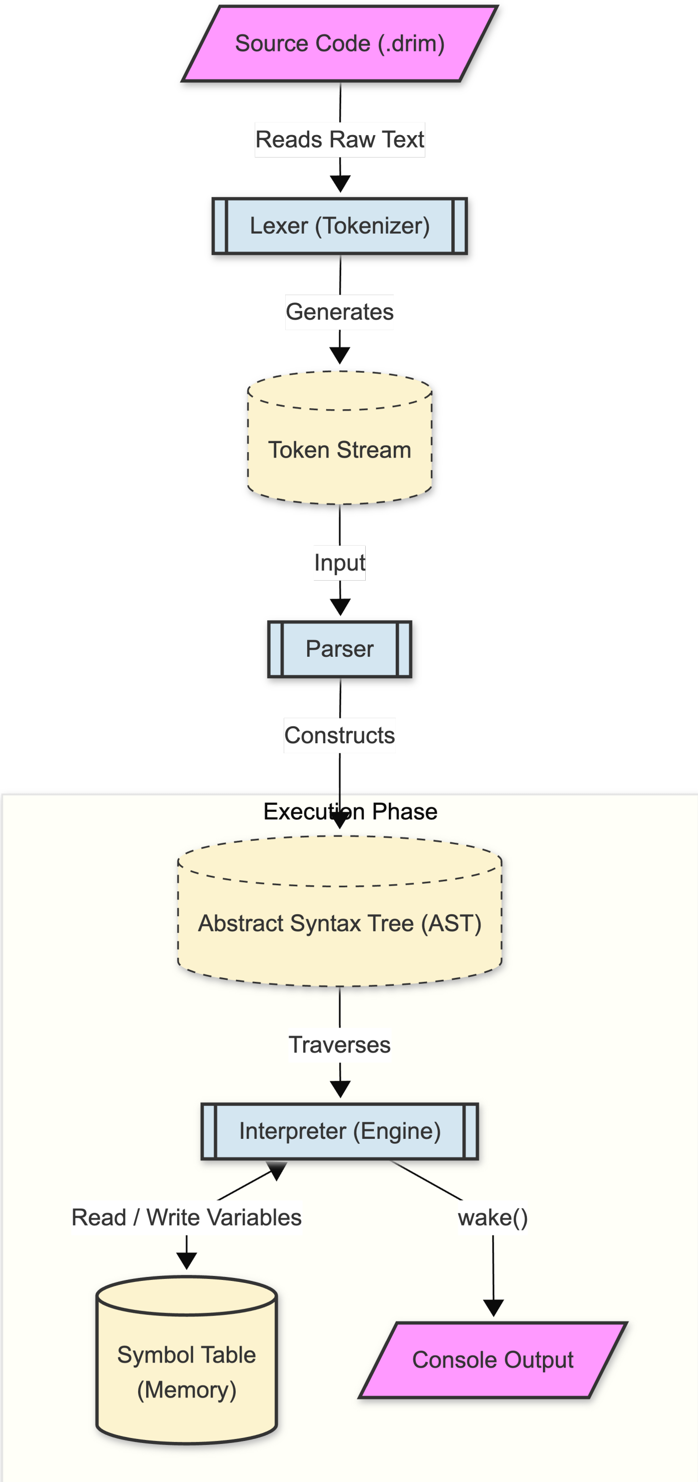

Figure 1: System Architecture Flow: From Source Code to Execution

Process Description

1. **Source Code (.drim):** The raw text file provided by the user containing the custom script.
2. **Lexer (Tokenizer):** Scans the source code character-by-character to generate a sequence of **Tokens** (Keywords, Identifiers, Literals).
3. **Parser:** Analyzes the token sequence against the language grammar rules. It constructs an **Abstract Syntax Tree (AST)** composed of Statement and Expression nodes.
4. **Interpreter:** Traverses the AST recursively. It executes Statements (actions) and evaluates Expressions (values), interacting with the **Environment** (Memory) to store and retrieve variable states.
5. **Output:** The result of the execution is printed to the standard output (Console).

6 Tools and Technologies

- **Programming Language:** C++ (Standard 17) - Chosen for performance and manual memory control.
- **Build System:** CMake - Used to manage the build process and ensure cross-platform compatibility.
- **Version Control:** Git & GitHub - Used for source code management, branching, and collaboration.
- **IDE:** CLion / Visual Studio Code - Primary development environments.
- **Diagramming:** Mermaid.js / Gamma - Used for creating Gantt charts and architecture diagrams.

7 Proposed Timeline

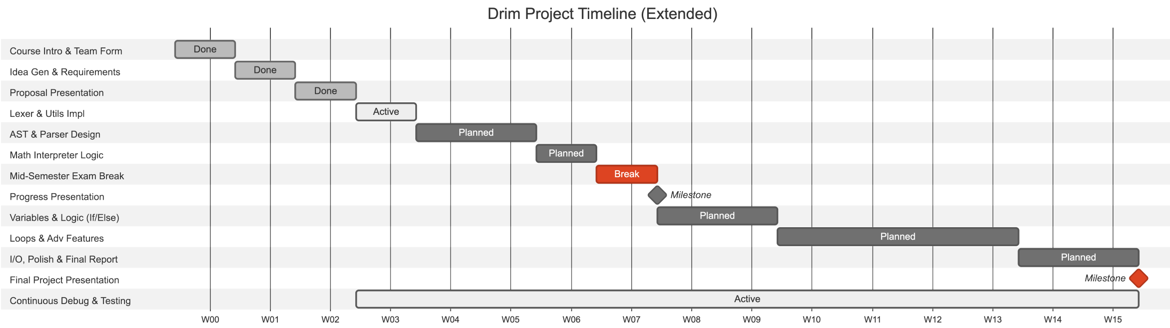

Figure 2: Timeline Diagram (Gantt Chart)

8 Suggestions Received

The suggestions that we received are -

- **Suggestion 1:** Details about suggestion 1 and how it will be addressed.
- **Suggestion 2:** Details about suggestion 2 and how it will be addressed.

9 Links

1. Presentation Slide Link
2. GitHub Repository Link