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 binaries, Drim functions as a **Tree-Walk Interpreter**. This means the application reads high-level source code from `.drim` files, parses it into an intermediate structure, and executes the logic in real-time within a custom runtime environment.

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 `wakeOut` for output), providing a simplified yet robust platform for logic execution.

The core of the project involves constructing the entire lifecycle of a programming language. This begins with a **Lexical Analyzer (Scanner)**, which breaks raw source text into meaningful tokens, categorizing them as identifiers, keywords, or literals. These tokens are then passed to a **Recursive Descent Parser**, which validates the syntax against a custom grammar and constructs a hierarchical **Abstract Syntax Tree (AST)**. This tree serves as the roadmap for the interpreter, representing the logical structure of the user's code.

This system manually manages memory allocation and undeclared variable declaration. The interpreter handles control flow dynamically, allowing for conditional branching and iteration. Furthermore, the system includes a robust error-reporting mechanism that identifies syntax violations or runtime anomalies (such as type mismatches or unexpected token) with precise location data.

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.

2 Motivation Behind the Project

At its core, Drim is about two things: preserving classroom culture and mastering complex engineering concepts.

Cultural Significance

The name **"Drim"** originated as an inside joke and a shared meme within the student batch from the very first day of our university life. Rather than letting this remain as a joke that might die over time, we decided to create something that will help us immortalize the memory of our classroom culture. Which is the reason we are personalizing this project with syntax and keywords like "DrimIn", "Drimming" to reflect our motivation. This relevance should increase our team's engagement and passion for the project, since there is a class full of people looking forward to seeing our project, materialization of our joke, unfold.

Engineering Challenges

From a technical perspective, this project serves as a demanding 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:** Restricting the use of third-party libraries and STL functions challenges us to engineer custom, innovative solutions from scratch and strengthens our problem-solving skills.
- **Collaboration and Project Management:** Working as a group on a complex software project enhances teamwork, version control practices, and project planning skills.

3 Key Features

The goal with Drim is to create something that balances simplicity with power. We are taking inspiration from standard C++ and Python architecture but deciding to go our own way with the implementation. By using custom keywords and limiting the feature set, we are avoiding unnecessary bloat. This will allow Drim to run logic through our custom Tree-Walk Interpreter implementation while keeping the code clean, readable, and uniquely ours.

- **Input/Output: Drim** accepts user input via the **drimIn()** command and displays formatted output using the **wakeOut()** command
- **Automatic Variable Assignment:** In the event of a user trying to assign a value to an undeclared variable, **DrimIn()** can declare the variable on its own with the intended data type and assign the value given by the user to the newly created variable.
- **Dynamic Type Casting:** Variables in **Drim** can hold either String or Integer or Boolean values, with implicit type casting handled by the interpreter. This feature simplifies variable management and enhances user experience.
- **Mathematical Operations: Drim** can correctly evaluate arithmetic expressions (+, -, *, /, %) respecting standard operator precedence.
- **Conversion Calculation:** Instead of requiring users to manually declare functions for common unit conversions, **Drim** provides pre-defined functions for converting between standard types (such as **degrees to radians**, **Fahrenheit to Celsius**, and **kilometers to feet**). This allows users to perform conversions efficiently without needing to look up rates externally.
- **Physics & Math Formulas: Drim** will be able to execute complex calculations, such as computing force or momentum, using pre-built functions rather than writing raw equations from scratch.
- **Control Flow:** The language supports conditional statements (`if-else`) and looping constructs (`drimming` loops) to let users control the flow of execution based on dynamic conditions.
- **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 `wakeOut()` 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 string printing, Drim supports advanced string interpolation. Users can embed variables directly within string literals (e.g., `wakeOut("Value: x))`, allowing for dynamic and readable output generation.
- **Helpful Error Reporting:** Coding rarely works perfectly on the first try, so **Drim** includes a robust error-reporting mechanism. It identifies syntax violations or runtime anomalies (such as type mismatches or unexpected token) and provides precise location data, helping users pinpoint exactly where things went wrong.

- **Headless Execution:** Designed specifically as a backend tool, Drim runs as a lightweight, headless Command Line Interface (CLI) application. It executes scripts fetched from a .drim file directly in the terminal without the overhead of a Graphical User Interface (GUI).
- **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, will be engineered manually using C++ 17.

4 Flow Chart/Class Diagram

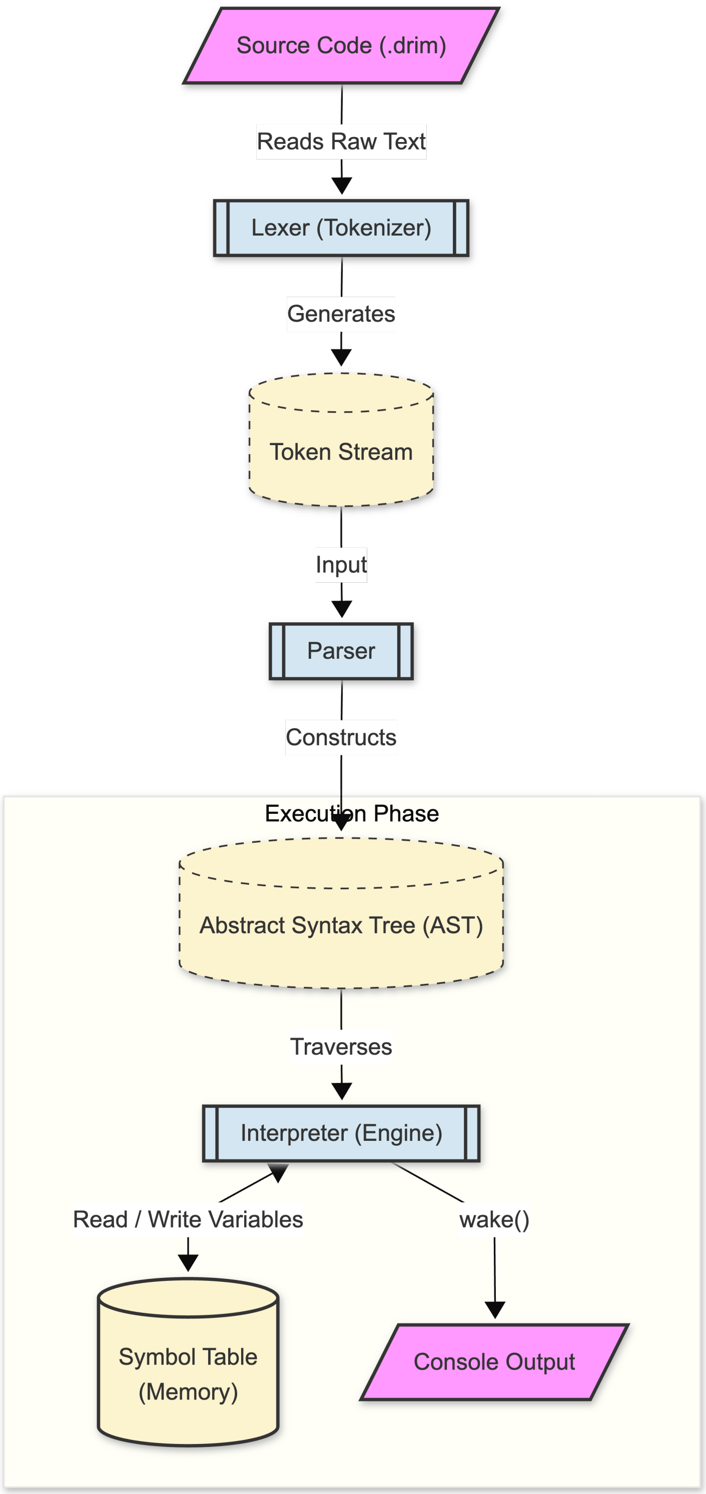

Figure 1: System Architecture Flow: From Source Code to Execution

Process Description

1. **Source Code (.drim):** The raw text file with .drim extension 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). These tokens are used to identify/tag if the word or character is an Identifier, or Keyword, or String Literal, or Assign Operator etc. These tokens are passed down to the parser for processing.

3. **Parser:** Parser analyzes the token sequence against the language grammar rules. It checks if the user passed any unexpected character sequence or syntax that does not make sense. If the source code makes sense grammatically, parser then constructs an **Abstract Syntax Tree (AST)** composed of Statement and Expression nodes.
   This AST is used to track which process needs to be calculated first. For Example: in wakeOut(2+3), instead of trying to print "2+3" as a whole (even though it can not since it is not any variable or string literal), the program will calculate the value of "2+3 = 5" and pass that integer to the wakeOut() function. Parser is where it builds that flow of execution.

4. **Interpreter:** The Interpreter is responsible for the actual execution of the code. It traverses the Abstract Syntax Tree recursively, treating it as a roadmap for logic. At each step, it evaluates Expressions to derive values—like turning that 2+3 node into a 5—and executes Statements to perform actions. It works hand-in-hand with the Environment, managing the scope and lifecycle of variables to ensure the program runs consistently from start to finish. The interpreter executes the lines according to the flow of execution set by the parser.

5. **Output:** The result of the execution is printed to the standard output (Console).

5 Tools and Technologies

In designing Drim, our tooling choices were driven by a philosophy of complete control without relying on ”black box” abstractions. Below are the technologies selected and the alternatives we weighed them against.

- **Programming Language: C++ (Standard 17)**

  Choice: We selected C++ 17 to leverage modern features while retaining the manual memory control required for a manually built interpreter.
  Alternatives: We considered **C++ 23** for its new library functions but opted out since usage of library functions is restricted.
  Managed languages like **Java** or **C#** were rejected to avoid the unpredictable performance overhead of Garbage Collection and the amount of library functions we need to use for basic tasks like input and output (scanner.nextInt(), scanner.nextLine()).
  We did not choose **Python** or **JavaScript** due to their interpreted nature, which would complicate low-level memory management and pointer manipulation.

- **Build System: CMake**

  Choice: Utilized to abstract the build process, ensuring the project allows for cross-platform compatibility without rewriting makefiles.
  Alternatives: GNU Make was considered but seemed unmanageable as the project directory structure will grow.
  Manual compilation scripts were rejected due to the difficulty of tracking header dependencies.

- **Version Control: Git & GitHub**

  Choice: Used for distributed and shared source code management, allowing us to implement experimental features in isolation using branches. And all team members are familiar with GitHub’s interface.
  Alternatives: Centralized systems like **SVN** were dismissed in favor of Git’s standard branching and merging capabilities, which facilitate better team collaboration.

- **IDE: CLion / Visual Studio Code**

  Choice: CLion is our primary environment for its in-depth memory inspection tools (vital for debugging segmentation faults), while **Visual Studio Code** is used as a lightweight editor for quick iteration.

- **Diagramming: Mermaid.js / Gamma**

  Choice: Used for creating Gantt charts and architecture diagrams.
  Alternatives: We avoided drag-and-drop tools like **Visio** or **Lucidchart** in favor of **Mermaid.js**, which allows us to treat ”diagrams as code” and version control our documentation alongside the source.

6 Proposed Timeline

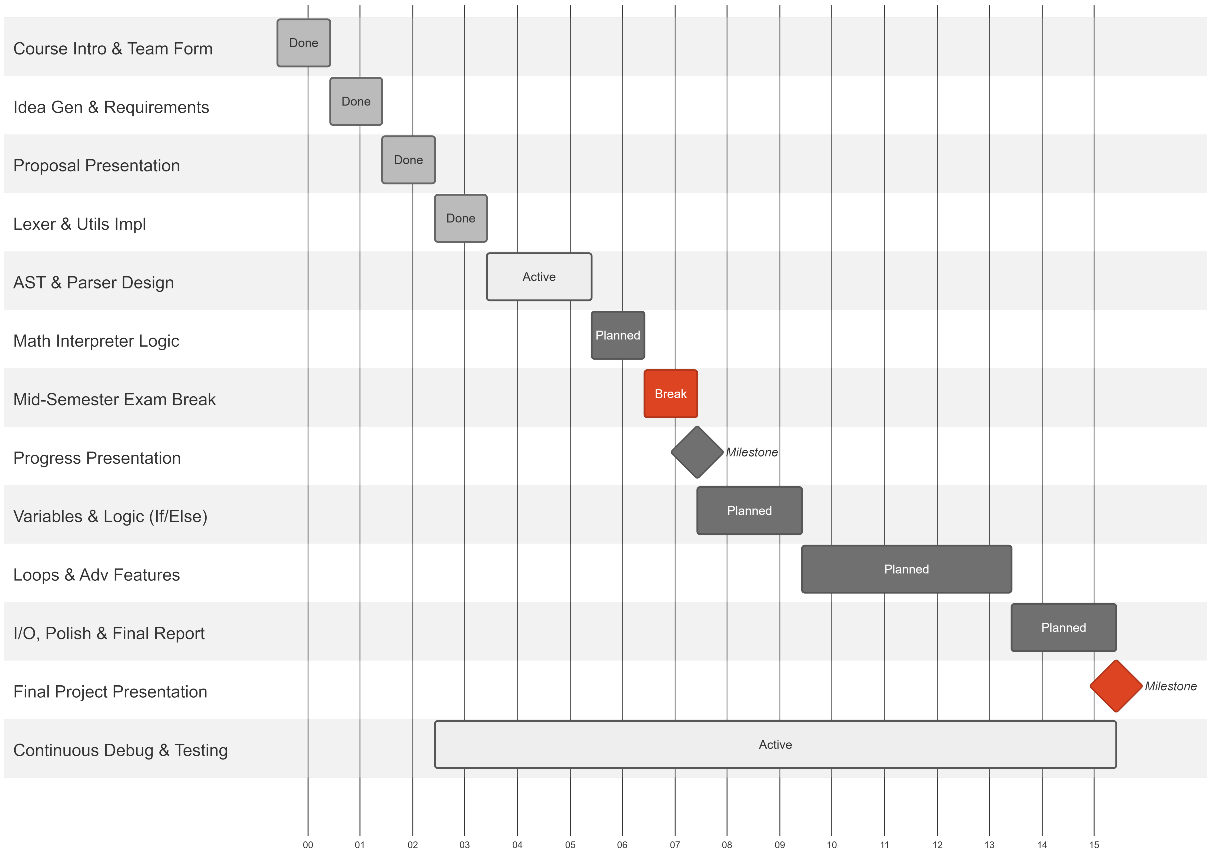

Figure 2: Timeline Diagram (Gantt Chart)

7 Suggestions Received

During our initial review, we received valuable feedback regarding both our development methodology and our project demonstration. Below are the specific suggestions and our roadmap to address them.

- **Development Strategy: Iterative Implementation**

  Suggestion: We were advised to adopt a ”small-to-big” approach, focusing on implementing small, functional components before attempting large-scale features.

  Resolution: We are developing basic arithmetic and variable assignment first. Only after these core mechanics are stable will we integrate complex modules like the Physics and Unit Conversion engines.

- **Presentation Quality: Rehearsal and Delivery**

  Suggestion: It was noted that our presentation delivery lacked polish, and we were advised to dedicate more time to practicing practice before future presentations.

  Resolution: We acknowledge that our previous delivery was not up to standard and our expectation. For the final presentation, the team will schedule multiple dry runs to ensure clear explanation of technical concepts and a smoother transition between speakers.

- **Time Management during Demonstrations**

  Suggestion: We were critiqued on our time management, as we struggled to cover all necessary material and finish our presentation within the allotted timeframe.

  Resolution: To prevent rushing or running overtime, we will structure our next presentation with a strict minute-by-minute agenda. We will allocate fixed time slots for the introduction, technical deep-dive, and live demo to ensure a balanced and comprehensive showcase.

8 Links

1. Presentation Slide Link
2. GitHub Repository Link