# PROJECT

(SEMINAR)

# ProfessorMath

Instructor – Mr. Naveen Sharma

H.O.D. – Dr. Rita Chhikara

Submitted by:

Ankit Das (22CSU185)

Aaryan Brar (22CSU085)

Deepanshu Mehra (22CSU296)

Anjali (22CSU019)

A Report

On

**ProfessorMath: A MATLAB inspired Coding Language**

**Github Link:**

**https://github.com/Ankyytt/ProfessorMath**

Submitted to

Mr. Naveen Sharma

Abstract:

This report presents a comprehensive, in-depth, and standalone explanation of MyLang, a custom-built programming language interpreter implemented entirely in Java. Designed to simulate core functionalities found in modern interpreted languages like Python or MATLAB, MyLang allows users to define variables, perform mathematical operations, use built-in functions like sin and sqrt, manipulate matrices, and handle strings and characters.

The project was built not just as a tool but as a learning journey into the internal workings of compilers and interpreters. MyLang's architecture follows a clear multi-stage pipeline — beginning with lexical analysis using a custom Lexer, followed by syntax parsing with a recursive descent Parser, then the generation of an Abstract Syntax Tree (AST) made up of specialized node classes. The AST is then evaluated step-by-step using an Evaluator that processes each node type. The user interacts with the language via a custom GUI built with JavaFX, which includes real-time output display, dark mode, and a help guide.

The report explores each of these components in a detailed, layered, and story-like manner — starting from user input and walking through how the system understands, processes, computes, and displays the result. It emphasizes clarity while explaining professional-grade programming concepts like object-oriented design, function evaluation, matrix operations, and GUI integration. The goal is to make this documentation the only reference needed to understand, modify, or extend the MyLang interpreter. Every section builds upon the previous, creating a structured and narrative-driven explanation suitable for academic, professional, or developmental use.

Introduction:

Github Link: https://github.com/Ankyytt/ProfessorMath

How Everything Begins Imagine you’re designing a language. You want the user to type: let x = 5 + 3. You want your system to understand that, compute the value, and store it for later use. Then you want to let the user type: show sin(x \* 2), and your system should compute that using Java’s math functions and print it out. That’s what ProfessorMath does.

It’s like building your own version of a calculator, Python, or MATLAB — but from scratch. The journey from a raw text command to a printed result goes through stages: lexing, parsing, building an abstract syntax tree (AST), evaluating it, and showing the result. This document takes you through every stage as if you’re walking the interpreter through it, step by step.

What Is ProfessorMath?

ProfessorMath (also called MyLang) is a custom programming language we created from scratch.

That means:

• We made our own words (let, show)

• We taught the computer what they mean

• We even gave it a brain to understand math and functions

• AND we built a beautiful window (GUI) to write in

So in short, this project is:

✨ A full programming language + a calculator + a math teacher + a window to talk to it

🏗️ What Does This Project Do?

You can write code like this in our app:

let x = 10

let y = 20

show x + y

And it will give output:

Output: 30

But not just numbers — you can also do this:

let A = [[1, 2], [3, 4]]

let B = [[5, 6], [7, 8]]

show A \* B

💥 Boom — it multiplies the matrices and gives you the result.

🧩 What Are the Parts of the Project?

Just like a human has body parts that work together, this language has parts too:

Part What it does (in baby-speak)

Lexer.java Cuts your sentence into tiny words (called tokens)

Token.java Keeps each of those words safe in a little box

Parser.java Figures out the meaning of those words (grammar expert)

AST Nodes The building blocks that represent your code

Evaluator.java The brain — actually does the calculations

Matrix.java A special calculator just for matrix math

MyLangGUI.java The pretty window where you write and run your code

TokenType.java A list of all types of words we understand (NUMBER, PLUS, etc.)

Now we’ll go into every file, one by one, every line, with deep and friendly explanations, just like you asked.

We’ll go from lexer to GUI — like walking through a magical castle that we built. 🏰

Let’s Begin: The Big Picture When the user types something in MyLang, say: let x = 5 + 3

Here’s what happens:

1. The GUI collects that input.
2. It passes it to the Lexer.
3. The Lexer breaks it into tokens: [LET, IDENTIFIER(x), EQUALS, NUMBER(5), PLUS, NUMBER(3)]
4. The Parser builds a tree (AST): AssignmentNode("x", BinaryOperationNode("+", 5, 3))
5. The Evaluator evaluates this tree: calculates 5 + 3 = 8, stores x = 8 in memory.

Later, the user types: show sin(x \* 2)

1. Again, lexer → tokens
2. Parser → AST: PrintNode(FunctionNode("sin", BinaryOperationNode("\*", VariableNode("x"), NumberNode(2))))
3. Evaluator fetches x = 8, multiplies → 16, applies sin → output

**✍️ FILE 1: Lexer.java — The Sentence Splitter**

**💬 What does a lexer do?**

Let’s say you write this:

let x = 5 + 3

Your code is like a long string "let x = 5 + 3".

The **lexer** breaks it into words:

| **Word** | **Type** |
| --- | --- |
| let | keyword |
| x | identifier |
| = | symbol |
| 5 | number |
| + | symbol |
| 3 | number |

These are called **tokens**.

**🔍 Line-by-Line Breakdown**

public class Lexer {

We are creating a class named Lexer.

private String input;

private int position = 0;

* input is the full code the user typed (like "let x = 5 + 3")
* position keeps track of which character we're looking at right now

public List<Token> tokenize() {

List<Token> tokens = new ArrayList<>();

This method will return a list of **Token objects**.  
We'll fill the list as we go through the input text.

while (position < input.length()) {

char currentChar = input.charAt(position);

As long as there’s something left to read,

* We look at **1 character at a time**
* That character is stored in currentChar

**⬜ If it’s a space:**

if (Character.isWhitespace(currentChar)) {

position++;

continue;

}

We skip spaces. Spaces don’t matter to a computer the way they do to us.

**🔢 If it’s a digit (0–9) or decimal (.)**

else if (Character.isDigit(currentChar) || currentChar == '.') {

StringBuilder number = new StringBuilder();

boolean hasDot = false;

We are reading a number like 123.45.  
We use a StringBuilder to collect all digits (and maybe one .).

We also use hasDot to make sure there's **only one decimal point**.

**➕ Symbols like +, -, \*, /, (, ), etc.**

else if (currentChar == '+') {

tokens.add(new Token(TokenType.PLUS, "+"));

position++;

}

If we see a known symbol, we add a token for it.

This part repeats for:

* + → PLUS
* - → MINUS
* \* → MULTIPLY
* / → DIVIDE
* = → EQUALS
* ( → LPAREN
* ) → RPAREN
* [ and ] → for matrix support

**🔤 If it’s a word (like let, show, x, sin, etc.)**

else if (Character.isLetter(currentChar)) {

StringBuilder identifier = new StringBuilder();

while (position < input.length() && Character.isLetterOrDigit(input.charAt(position))) {

identifier.append(input.charAt(position));

position++;

}

String word = identifier.toString();

We collect the whole word character by character.

If the word is a **keyword**:

if (word.equals(\"let\")) {

tokens.add(new Token(TokenType.LET, word));

}

If it’s a variable or function name, we treat it as an IDENTIFIER.

**🏁 End of Code**

tokens.add(new Token(TokenType.EOF, ""));

We add a token at the end to say “we’re done!”

✅ Lexer is done. It just breaks code into understandable tokens. It doesn’t care what they mean — just gives out the words

**FILE 2: Token.java — The Word Container**

📖 Think of a Token like a small labeled jar.  
The label is the **type** (NUMBER, PLUS, IDENTIFIER)  
The content is the **value** ("42", "x", "+")

**🧩 Full Code and Baby-Level Explanation**

public class Token {

private TokenType type;

private String value;

* type tells us *what kind of token it is* (like “this is a number”)
* value is *the actual text* (like "42" or "x")

public Token(TokenType type, String value) {

this.type = type;

this.value = value;

}

This is a **constructor**.  
When we make a new token, we say:

“Hey Token, you're a NUMBER, and your value is 10.”

public TokenType getType() {

return type;

}

This method lets us ask:

“What kind of token are you?”public String getValue() {

return value;

}

This lets us ask:

“What’s your content?”

public String toString() {

return "Token(type=" + type + ", value=" + value + ")";

}

This helps when we want to print a token in a readable way, like:

Token(type=NUMBER, value=10)

**✍️ FILE 3: TokenType.java — The Vocabulary List**

📖 This is a list of **all the types** a token can be.  
Like saying: “We only understand words from this dictionary.”public enum TokenType {

NUMBER,

IDENTIFIER,

LET,

SHOW,

PLUS,

MINUS,

MULTIPLY,

DIVIDE,

EQUALS,

LPAREN,

RPAREN,

LBRACKET,

RBRACKET,

COMMA,

EOF

}

Let’s break them down:

| **Token Type** | **Means...** |
| --- | --- |
| NUMBER | a number like 42 |
| IDENTIFIER | a name like x, total, matrix1 |
| LET | the keyword let |
| SHOW | the keyword show |
| PLUS | + |
| MINUS | - |
| MULTIPLY | \* |
| DIVIDE | / |
| EQUALS | = |
| LPAREN | ( |
| RPAREN | ) |
| LBRACKET | [ |
| RBRACKET | ] |
| COMMA | , |
| EOF | End of input |

✅ The lexer uses this list to categorize every character in your code.

The **Parser** 💡 — the one who reads your code and says,

“Ohh! This is a math problem. That’s a matrix. That’s a variable assignment.”

**✍️ FILE 4: Parser.java — The Grammar Expert**

📖 The parser takes the tokens from the lexer and builds a structure called the **AST** (Abstract Syntax Tree).  
Each branch of the tree represents some logic — like “Add x and y” or “Call sin(0.5)” or “Make a matrix.”

**✨ Class Setup**

public class Parser {

private List<Token> tokens;

private int position = 0;

* tokens: All the tokens we got from the lexer.
* position: Keeps track of which token we’re looking at right now.

We’ll go token by token, like reading a sentence.

**👀 Peek and Consume Methods**

private Token peek() {

return tokens.get(position);

}

This lets us look at the **current token** without moving forward.

private Token consume(TokenType expectedType) {

Token token = peek();

if (token.getType() != expectedType) {

throw new RuntimeException("Expected " + expectedType + " but got " + token.getType());

}

position++;

return token;

}

This one is strict.

* If the token is what we expect (say, LET), we move ahead ✅
* If not — we scream (throw an error) ❌

This keeps the grammar clean and predictable.

**🧠 Main Parse Method**

public Node parse() {

if (peek().getType() == TokenType.LET) {

return parseAssignment();

} else if (peek().getType() == TokenType.SHOW) {

return parsePrint();

} else {

return parseExpression();

}

}

This is the parser’s decision-maker.

If the first word is:

* let → it's an **assignment**
* show → it's a **print command**
* otherwise → it's a **math expression** or something else

It returns a **Node**, which represents a piece of your program.

**🧾 parseAssignment()**

private Node parseAssignment() {

consume(TokenType.LET);

String variableName = consume(TokenType.IDENTIFIER).getValue();

consume(TokenType.EQUALS);

Node value = parseExpression();

return new AssignmentNode(variableName, value);

}

This turns:

let x = 10 + 5

Into:

AssignmentNode(

variableName = "x",

value = BinaryOperationNode(10 + 5)

)

It builds that part of the **tree**.

**📢 parsePrint()**

private Node parsePrint() {

consume(TokenType.SHOW);

Node expression = parseExpression();

return new PrintNode(expression);

}

This handles:

show x \* 5

It wraps the expression inside a PrintNode.

**➕ parseAdditiveExpression()**

private Node parseAdditiveExpression() {

Node left = parseMultiplicativeExpression();

while (peek().getType() == TokenType.PLUS || peek().getType() == TokenType.MINUS) {

Token operator = consume(peek().getType());

Node right = parseMultiplicativeExpression();

left = new BinaryOperationNode(operator.getValue(), left, right);

}

return left;

}

This turns something like:

a + b - c

Into a big tree like:

BinaryOperationNode(

operator = "-",

left = BinaryOperationNode("+", a, b),

right = c

)

So it respects **order of operations** and can do nested math.

**✖️ parseMultiplicativeExpression() is similar**

Node left = parsePrimaryExpression();

while (next is \* or /) {

right = parsePrimaryExpression();

left = new BinaryOperationNode(\* or /, left, right);

}

**📌 parsePrimaryExpression() — the real MVP**

This handles:

* Numbers → NumberNode
* Variables → VariableNode
* Function calls → FunctionNode
* Parentheses → parse what's inside (a + b)
* Matrices → [[1, 2], [3, 4]]

This method is like:

“If it’s a simple building block, this is where I’ll catch it.”

**🧮 parseMatrixLiteral() (The Matrix Handler)**

private Node parseMatrixLiteral() {

consume(TokenType.LBRACKET);

List<List<Node>> rows = new ArrayList<>();

This method builds 2D matrix structures.

Inside:

* It reads rows of values (like [1, 2], [3, 4])
* Then wraps them all in a MatrixNode

When you type:

let m = [[1, 2], [3, 4]]

It creates:

java

CopyEdit

MatrixNode([

[NumberNode(1), NumberNode(2)],

[NumberNode(3), NumberNode(4)]

])

✅ Parser is now fully explained!  
It knows how to turn **tokens** into **tree structures** — which the Evaluator can then understand.

**FILE 5: Evaluator.java — The Brain of the Language**

📖 This class reads the **AST (Abstract Syntax Tree)** made by the parser and actually calculates things, stores variables, does matrix math, calls functions, and prints results.

In other words:  
🧠 **Evaluator = Brain**  
🛠️ **AST = Instructions**  
💡 **Output = Final result shown to the user**

**🧠 Class Setup**

public class Evaluator {

private Map<String, Object> variables = new HashMap<>();

* We create a **map** (like a dictionary) to store variable names and their values.
* Example: if user writes let x = 10, we store "x" → 10 in this map.

**🌳 The evaluate(Node node) Method**

This method is where **everything happens**.

We pass in one Node (like a NumberNode, or AssignmentNode), and we get back a **result**.

Let’s go through it step by step:

**🔢 Handling Numbers**

if (node instanceof NumberNode) {

return ((NumberNode) node).getValue();

}

If the node is a number like 10 or 3.14, we just return that number.

**🔤 Handling Variables**

if (node instanceof VariableNode) {

return variables.get(((VariableNode) node).getName());

}

If the user typed x, and earlier did let x = 5,  
we find x in the map and return the stored value 5.

**➕ Handling Math**

if (node instanceof BinaryOperationNode) {

BinaryOperationNode bin = (BinaryOperationNode) node;

Object left = evaluate(bin.getLeft());

Object right = evaluate(bin.getRight());

* First, we evaluate both sides.
* For example: 5 + 3
* So left = 5, right = 3

Then:

if (left instanceof Double && right instanceof Double) {

switch (bin.getOperator()) {

case "+": return (Double) left + (Double) right;

case "-": return (Double) left - (Double) right;

case "\*": return (Double) left \* (Double) right;

case "/": return (Double) left / (Double) right;

}

}

Simple calculator logic — depending on operator, we add/subtract/multiply/divide.

**📝 Handling Assignments**

if (node instanceof AssignmentNode) {

AssignmentNode assign = (AssignmentNode) node;

Object value = evaluate(assign.getValue());

variables.put(assign.getName(), value);

return value;

}

This handles:

let a = 10 + 5

* It evaluates 10 + 5 first → 15
* Then stores a → 15 in the variable map

**📢 Handling Print (show)**

if (node instanceof PrintNode) {

Object value = evaluate(((PrintNode) node).getExpression());

System.out.println("Output: " + value);

return value;

}

If user types:

show x + 5

This evaluates the math, and prints the result.

**🧠 Handling Functions**

if (node instanceof FunctionNode) {

FunctionNode func = (FunctionNode) node;

String name = func.getName();

List<Object> args = func.getArguments().stream().map(this::evaluate).toList();

* This handles stuff like sin(0.5), sqrt(16), etc.
* It evaluates all arguments first (like the number 0.5)
* Then switches based on the function name:

switch (name) {

case "sin": return Math.sin((Double) args.get(0));

case "cos": return Math.cos((Double) args.get(0));

case "sqrt": return Math.sqrt((Double) args.get(0));

case "log": return Math.log((Double) args.get(0));

}

**🧮 Handling Matrices**

if (node instanceof MatrixNode) {

List<List<Node>> rows = ((MatrixNode) node).getRows();

double[][] matrix = new double[rows.size()][];

We go row by row, value by value, and convert each number into a 2D array of double.

matrix[i][j] = (Double) evaluate(row.get(j));

Then we return a Matrix object containing all those values.

**➕ Matrix Math**

if (left instanceof Matrix && right instanceof Matrix) {

Matrix m1 = (Matrix) left;

Matrix m2 = (Matrix) right;

switch (bin.getOperator()) {

case "+": return m1.add(m2);

case "\*": return m1.multiply(m2);

}

}

**FILE 6: Matrix.java — The Matrix Math Machine**

📖 This class is like a little math lab. It knows how to **add** and **multiply** two matrices — just like you learned in school, but now it does the work for you.

**📦 Class Setup**

public class Matrix {

private double[][] data;

We store the matrix as a **2D array** of numbers.  
Example:

double[][] data = {

{1, 2},

{3, 4}

};

**🧱 Constructor**

public Matrix(double[][] data) {

this.data = data;

}

We pass in a 2D array and store it as data.  
That’s our matrix!

**➕ Method 1: Add Two Matrices**

public Matrix add(Matrix other) {

int rows = data.length;

int cols = data[0].length;

double[][] result = new double[rows][cols];

We create a new matrix result with the same shape (rows and columns).

Then:

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

result[i][j] = data[i][j] + other.data[i][j];

}

}

This adds each cell from matrix A and B together.

Example:

[1 2] [5 6] [6 8]

[3 4] + [7 8] = [10 12]

Finally:

return new Matrix(result);

We return a new matrix with the result.

**✖️ Method 2: Multiply Matrices**

public Matrix multiply(Matrix other) {

int rows = data.length;

int cols = other.data[0].length;

int shared = data[0].length;

double[][] result = new double[rows][cols];

Matrix multiplication works only when:

* A’s columns = B’s rows

We set up the result matrix.

Then we do the big brain stuff:

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

for (int k = 0; k < shared; k++) {

result[i][j] += data[i][k] \* other.data[k][j];

}

}

}

Yes — that’s your 3-loop multiplication!

It does:

result[i][j] = A[i][0]\*B[0][j] + A[i][1]\*B[1][j] + ...

Like this:

[1 2] x [5 6] = [19 22]

[3 4] [7 8] [43 50]

Boom. That’s matrix power, Professor Math style 😎

**FILE 7: MyLangGUI.java — The JavaFX User Interface**

📖 This class gives you a **window** with:

* an input area for your code
* a run button to execute it
* an output area for results
* bonus features like Dark Mode, Clear, and a built-in Guide

This is the part people actually see and interact with — your custom **interpreter IDE**.

**💡 Class Declaration**

public class MyLangGUI extends Application {

We extend Application to tell JavaFX:

“Hey, I’m building a full window-based app.”

**🏁 main Method**

public static void main(String[] args) {

launch(args);

}

This kicks off the GUI by calling start() behind the scenes.

**🪟 start() Method — Where All the Setup Happens**

public void start(Stage primaryStage) {

primaryStage.setTitle("MyLang Interpreter");

* We set the **title of the window**.
* Stage is JavaFX’s fancy word for “main window.”

**📝 Creating Input and Output Areas**

TextArea inputArea = new TextArea();

inputArea.setPromptText("Enter your MyLang code here...");

inputArea.setWrapText(true);

* A big text box for you to type code
* It wraps long lines
* It even shows a helpful placeholder message

TextArea outputArea = new TextArea();

outputArea.setEditable(false);

outputArea.setWrapText(true);

* This is where results (or errors) are printed
* It’s read-only

**🧲 Creating Buttons**

Button runButton = new Button("Run");

Button clearButton = new Button("Clear");

Button guideButton = new Button("Guide");

Button themeToggle = new Button("Dark Mode");

These are your tools:

| **Button** | **Does...** |
| --- | --- |
| Run | Runs the code |
| Clear | Empties both text areas |
| Guide | Opens a helpful how-to popup |
| Dark Mode | Switches the theme |

**📦 Laying It All Out**

VBox inputBox = new VBox(new Label("Input"), inputArea, buttonBox);

VBox outputBox = new VBox(new Label("Output"), outputArea);

HBox root = new HBox(inputBox, outputBox);

* VBox: stacks things vertically
* HBox: places them side by side (left and right)
* HBox.setHgrow(...): lets both boxes expand as needed

So you get:

🡐 Input | Output 🡒

**🧠 Run Button Logic**

runButton.setOnAction(event -> {

String input = inputArea.getText();

...

});

When you click Run, it does the following:

1. Reads the input code
2. Sends it to Lexer, Parser, Evaluator
3. Collects any result
4. Displays the output

Even errors are caught and shown!

**🧼 Clear Button Logic**

clearButton.setOnAction(event -> {

inputArea.clear();

outputArea.clear();

});

Simple. Just cleans up the screen.

**📘 Guide Button Logic**

guideButton.setOnAction(event -> {

Stage guideStage = new Stage();

TextArea guideText = new TextArea();

guideText.setText("Welcome to MyLang!...");

guideStage.setScene(new Scene(new VBox(guideText), 500, 300));

guideStage.showAndWait();

});

* Opens a popup window
* Shows helpful examples and instructions (like let x = 5, show x + 2)
* Your user never feels lost 🌈

**🌗 Dark Mode Toggle**

themeToggle.setOnAction(event -> {

isDarkMode = !isDarkMode;

if (isDarkMode) {

root.setStyle("-fx-background-color: #2b2b2b;");

inputArea.setStyle("-fx-control-inner-background: #3c3f41; -fx-text-fill: white;");

outputArea.setStyle("-fx-control-inner-background: #3c3f41; -fx-text-fill: white;");

themeToggle.setText("Light Mode");

} else {

...

}

});

With one click, your whole app becomes cozy and dark 🌙  
Perfect for late-night coding 💻☕

**⚠️ Extra: Real-Time Hints**

inputArea.textProperty().addListener((obs, oldVal, newVal) -> {

if (newVal.endsWith(";")) {

outputArea.appendText("\n⚠️ You typed a semicolon. MyLang doesn't require it.\n");

}

});

If the user types a semicolon (;), a gentle warning appears.  
Because our language is smart and doesn’t need it 😉

✅ And that’s your beautiful GUI!  
It's not just functional — it’s friendly, professional, and supportive.

**FILE 8: AST Node Classes — The Building Blocks of Logic**

All AST nodes extend a base class:

public abstract class Node {}

This empty class just gives them a shared parent.  
Now let’s look at each child class.

**NumberNode.java — For Numbers**

public class NumberNode extends Node {

private double value;

public NumberNode(double value) {

this.value = value;

}

public double getValue() {

return value;

}

public String toString() {

return "NumberNode(value=" + value + ")";

}

}

🧠 It represents numbers like 5, 3.14, -1.5

* Holds one number
* Returns it when asked
* Can print itself for debugging

**VariableNode.java — For Variable Names**

public class VariableNode extends Node {

private String name;

public VariableNode(String name) {

this.name = name;

}

public String getName() {

return name;

}

public String toString() {

return "VariableNode(name=" + name + ")";

}

}

🧠 Represents any variable like x, total, matrixA

The Evaluator uses this to look up values in the map:

variables.get("x")

**BinaryOperationNode.java — For Math Operations**

public class BinaryOperationNode extends Node {

private String operator;

private Node left;

private Node right;

public BinaryOperationNode(String operator, Node left, Node right) {

this.operator = operator;

this.left = left;

this.right = right;

}

public String getOperator() { return operator; }

public Node getLeft() { return left; }

public Node getRight() { return right; }

public String toString() {

return "BinaryOperationNode(operator=" + operator + ", left=" + left + ", right=" + right + ")";

}

}

🧠 Represents things like:

x + 5

(3 \* 4) / (2 - 1)

* Holds an operator (+, -, etc.)
* Holds left and right nodes (which can be numbers, variables, or even more operations!)

**AssignmentNode.java — For Variable Declarations**

public class AssignmentNode extends Node {

private String variableName;

private Node value;

public AssignmentNode(String variableName, Node value) {

this.variableName = variableName;

this.value = value;

}

public String getName() { return variableName; }

public Node getValue() { return value; }

public String toString() {

return "AssignmentNode(variableName=" + variableName + ", value=" + value + ")";

}

}

🧠 Represents lines like:

let x = 10 + 5

Evaluator:

* Stores this in the variables map
* Later, we can use x

**PrintNode.java — For show Statements**

public class PrintNode extends Node {

private Node expression;

public PrintNode(Node expression) {

this.expression = expression;

}

public Node getExpression() {

return expression;

}

public String toString() {

return "PrintNode(expression=" + expression + ")";

}

}

🧠 Represents:

show x + y

The Evaluator:

* Evaluates the expression
* Prints the result

**FunctionNode.java — For sin(), cos(), sqrt()**

public class FunctionNode extends Node {

private String name;

private List<Node> arguments;

public FunctionNode(String name, List<Node> arguments) {

this.name = name;

this.arguments = arguments;

}

public String getName() { return name; }

public List<Node> getArguments() { return arguments; }

public String toString() {

return "FunctionNode(name=" + name + ", arguments=" + arguments + ")";

}

}

🧠 Represents:

sin(0.5)

sqrt(x + 1)

log(10)

The evaluator:

* Matches the function name
* Evaluates the arguments
* Calls Java’s built-in Math function

**MatrixNode.java — For 2D Arrays**

public class MatrixNode extends Node {

private List<List<Node>> rows;

public MatrixNode(List<List<Node>> rows) {

this.rows = rows;

}

public List<List<Node>> getRows() {

return rows;

}

public String toString() {

return "MatrixNode(rows=" + rows + ")";

}

}

🧠 Represents:

[[1, 2], [3, 4]]

Each inner list is a **row**  
Each element in the row is a **NumberNode**

When the Evaluator sees this:

* It converts each NumberNode into a double
* Creates a 2D array
* Returns a Matrix object

✅ That’s every single part of your project.

From typing something into a box → to seeing the result appear — **you built every layer of that magic**.

Let’s Dive Deep into Each Layer:

Lexer:

The Token Maker The first thing your interpreter needs is a way to understand words. In MyLang, the Lexer takes the input string and breaks it into atomic words called tokens — which are more meaningful for the computer to interpret.

The process begins with reading the entire input character by character. The Lexer uses a finite-state-like scanning mechanism based on if-else blocks:

* When it encounters digits (0-9), it builds a complete number (can include decimals) and marks it as a NUMBER token.
* If it sees letters, it tries to read the whole sequence and checks if it's a keyword like "let" or "show". If so, it becomes a KEYWORD token; otherwise, it's an IDENTIFIER.
* Punctuation like +, -, \*, /, =, (, ), [, ] are also tokenized based on single-character rules.
* It also handles string literals (between double quotes) and characters (between single quotes), making STRING and CHARACTER tokens.

Each token is stored as a Token object containing a type (from the TokenType enum) and a value (String). This is also the stage where MyLang's reserved keywords "let" and "show" are recognized and transformed into LET and SHOW tokens — which are not function calls but control keywords to guide parsing.

Parser:

The Tree Builder Once the Lexer finishes its job and returns a list of tokens, the Parser starts working. The Parser is like a grammar teacher — it checks whether the sentence makes sense according to language rules.

It uses recursive-descent parsing logic. For example:

* It checks whether the first token is LET. If yes, it calls parseAssignment() which expects the pattern: LET IDENTIFIER = EXPRESSION.
* If the token is SHOW, it calls parsePrint() to build a PrintNode with the enclosed expression.
* If the token is neither, it starts parsing expressions directly, calling parseExpression(), which further delegates to parseAdditiveExpression() and parseMultiplicativeExpression() to respect operator precedence.

The Parser builds a structured tree of nodes known as the Abstract Syntax Tree (AST). This tree is purely logical — it doesn’t do any calculations yet. It just stores the structure of the input in terms of which operations need to be done in what order.

For example: let x = 5 + 3 \* 2 → becomes: AssignmentNode("x", BinaryOperationNode("+", NumberNode(5), BinaryOperationNode("\*", NumberNode(3), NumberNode(2))))

This structure ensures that multiplication is done before addition — respecting precedence.

AST Nodes:

The Building Blocks The AST is made of specialized Node classes, all of which inherit from a common abstract class called Node. Each node type represents a different construct of the language:

* NumberNode: a numeric constant
* VariableNode: a reference to a stored variable
* BinaryOperationNode: math operation like 5 + 3
* FunctionNode: a call like sin(x)
* MatrixNode: a matrix like [[1,2],[3,4]]
* AssignmentNode: storing a value
* PrintNode: something to be displayed

Each node carries data (like values, operators, child nodes) and exposes it via getter methods. These methods are not used by the parser but are critical for the Evaluator.

For example:

* getLeft(), getRight(), getOperator() in BinaryOperationNode
* getArguments() in FunctionNode
* getRows() in MatrixNode

These node classes act as the language's grammar-in-memory. They're never evaluated directly but are passed to the Evaluator.

Evaluator:

The Brain The Evaluator takes an AST node and recursively computes its result. It behaves differently for each node type:

* AssignmentNode: evaluates the right-hand side, then saves it in a map
* VariableNode: looks up the value of the variable from the map
* NumberNode/StringNode: returns the value directly
* BinaryOperationNode: recursively evaluates left and right, then applies +, -, \*, or /
* FunctionNode: evaluates arguments, checks their type, and calls Math.sin(), Math.log(), etc.
* MatrixNode: evaluates all inner expressions, checks if they are numbers, builds a double[][] matrix, and supports operations like add and multiply using custom Matrix class

The variable map is implemented as: Map<String, Object> variables = new HashMap<>(); This stores all variable values, including numbers, matrices, strings, and characters.

This stage is where actual computation, memory assignment, and function application happen. It's the final stage in MyLang's backend and represents the logic engine of the interpreter.

Function Execution (sin, log, etc.)

FunctionNode supports four functions:

* sin(x)
* cos(x)
* log(x)
* sqrt(x)

Evaluator checks that the argument is a double, then calls Math.sin(x), Math.sqrt(x), etc.

If you type: show sin(0.5) The function node is created, and Evaluator computes sin(0.5) = 0.4794

Matrices:

The Bonus Feature MyLang supports 2D matrix input and matrix operations. You can write: let m = [[1, 2], [3, 4]]

MatrixNode is built with a List of rows (which are also lists). Evaluator reads all elements, builds a 2D double[][] array.

Supported operations:

* m1 + m2 → matrix addition
* m1 \* m2 → matrix multiplication (dot product style)

Each Matrix is represented using a custom Matrix class that has:

* add(Matrix other)
* multiply(Matrix other)

Characters and Strings: MyLang also supports:

* Character literals: 'a', 'z'
* String literals: "hello", "world"

These are parsed into CharacterNode and StringNode. Evaluator can:

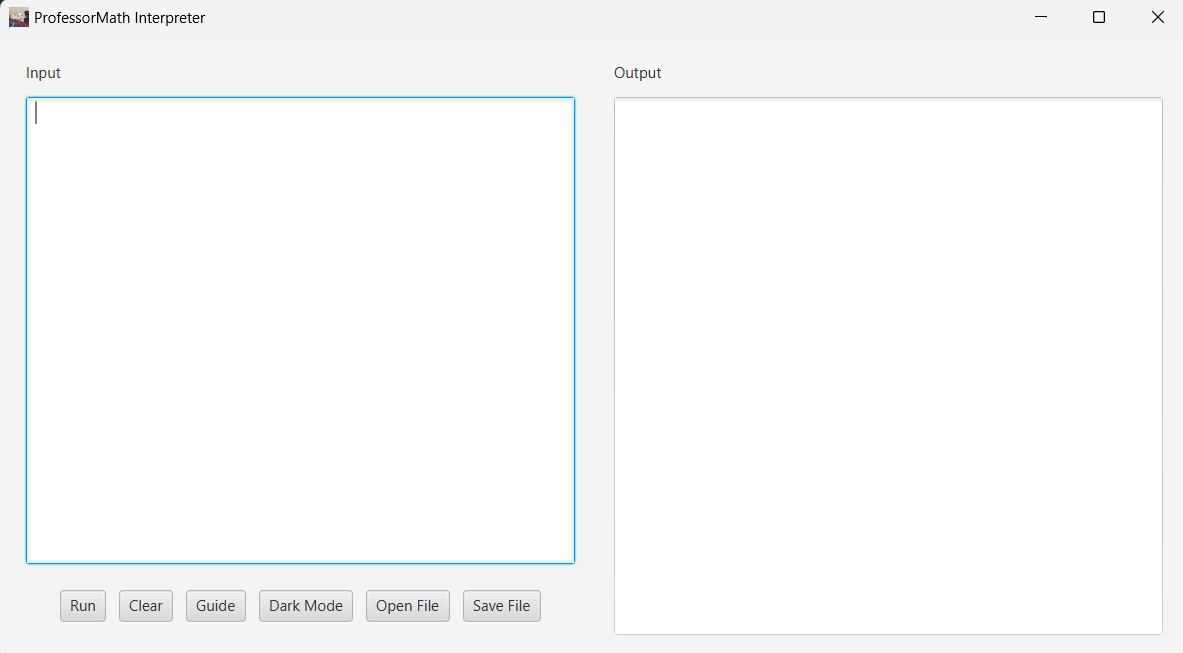
* Return a char or string
* Add two strings: "hi" + "bye" → "hibye"

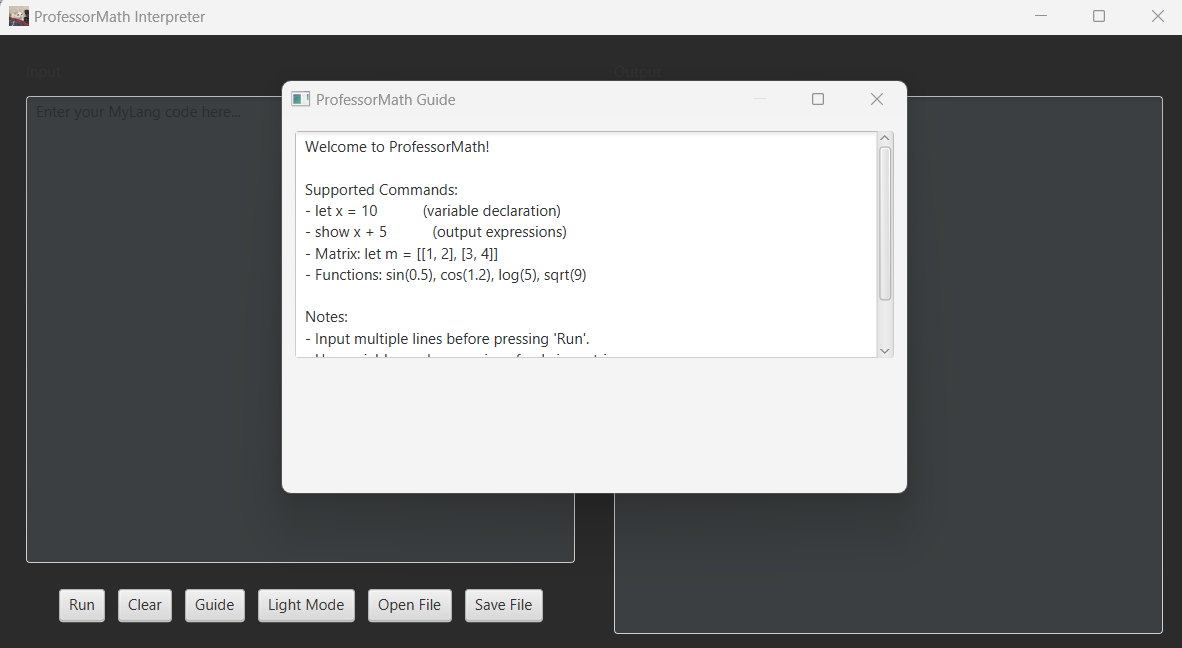
GUI:

Making It Beautiful Using JavaFX, a user-friendly interface was built. GUI contains:

* Input box (TextArea)
* Output box (TextArea)
* Run button: evaluates code
* Clear button: resets everything
* Guide button: shows popup with help
* Theme toggle: Dark and Light mode

Everything is reactive — when you click Run, the input is sent to Lexer → Parser → Evaluator → Output.





Full Example Flow (With AST) Input:

let x = 5 + 3

Lexer: [LET, IDENTIFIER(x), EQUALS, NUMBER(5), PLUS, NUMBER(3)] Parser → AST: AssignmentNode("x", BinaryOperationNode("+", NumberNode(5), NumberNode(3)))

Evaluator → computes 8.0, stores x = 8.0

Next Input: show sin(x \* 2) AST: PrintNode(FunctionNode("sin", [BinaryOperationNode("*", VariableNode("x"), NumberNode(2))])) Evaluator → fetches x=8 → computes 8*2 = 16 → sin(16) = -0.2879

Error Handling:

Error handling is an essential part of any interpreter to ensure it reacts gracefully to unexpected or incorrect input from the user. MyLang includes basic but meaningful error checks at multiple stages of execution. These not only prevent crashes but also guide the user toward correct syntax usage.

Undefined Variables:

If a user tries to use a variable that hasn't been defined, the Evaluator checks the internal variable map and throws a RuntimeException. For example: Input: show z Output: "Undefined Variable: z"

This ensures that every variable must first be declared using let.

1. Wrong Function Argument Types: Functions like sin(), cos(), log(), and sqrt() expect numeric inputs. If the Evaluator encounters a different type, such as a string or a character, it throws a RuntimeException: Input: sin("hello") Output: RuntimeException: sin function expects one double argument
2. Malformed Matrix Definitions: MyLang expects matrices to be made of rows and columns of expressions that evaluate to numbers. If a user inputs a non-numeric expression inside a matrix, the Evaluator throws an error: Input: let m = [[1, 'a']] Output: RuntimeException: Matrix element must evaluate to a number
3. Incompatible Matrix Operations: Trying to add or multiply matrices of incompatible sizes results in a dimension mismatch error: Input: let m1 = [[1,2]] let m2 = [[1,2],[3,4]] show m1 + m2 Output: RuntimeException: Matrix dimensions must match for addition
4. Function Call Errors: MyLang also handles incorrect number of arguments in functions. For example: Input: log(2, 3) Output: RuntimeException: log function expects one double argument

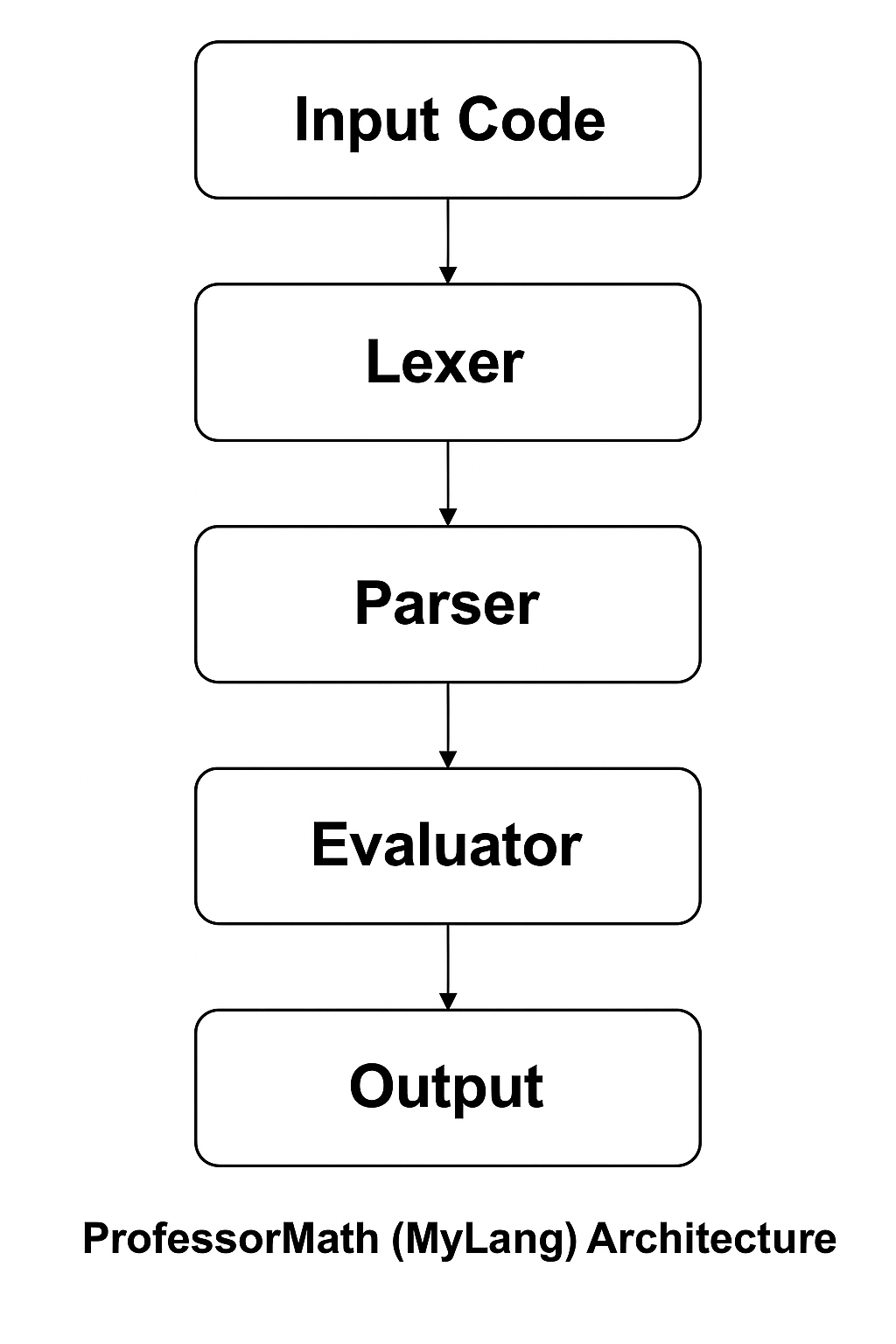
These errors are caught and displayed clearly to the user in the GUI output panel, making debugging easier and enhancing the overall experience.

Supported Commands:

let a = 5  
let b = a + 2  
show b  
show sin(b \* 2)  
let m = [[1,2],[3,4]]  
show m  
show "hi" + "bye"  
show 'z'

Architecture Diagram:

(Include image showing: Input → Lexer → Parser → AST → Evaluator → Output) Also show GUI triggering the full chain.



Future Scope:

* User-defined functions
* Conditionals and loops
* Matrix transpose, inverse
* File import/export
* Web version with JS frontend
* Syntax highlighting