Mini Project Report

On

"MyCode: Creating a Language from Scratch"

Submitted for partial fulfillment of the degree of

BACHELOR OF ENGINEERING

(Information Technology)

By

Mr. Ayush Waghade (A-10)

Mr. Gaurav Kotecha (A-17)

Mr. Atharv Dharmale (A-09)

Mr. Charudatta Mohod (A-12)

Semester: VI, Year: III

Under the guidance of

Dr. Pranjali P. Deshmukh

Department of Information Technology,

Prof. Ram Meghe Institute of Technology & Research, Badnera.

Year 2024-25

Prof. Ram Meghe Institute of Technology & Research, Badnera.

Department of Information Technology,

Year 2024-25

Certificate

This is to certify that the project report entitled

"MyCode: Creating a Language from Scratch"

is a bonafide work and it is submitted to the Sant Gadge Baba Amravati University, Amravati

By

Mr. Ayush Waghade (A-10)

Mr. Gaurav Kotecha (A-17)

Mr. Atharv Dharmale (A-09)

Mr. Charudatta Mohod (A-12)

in the partial fulfillment of the requirement for the award of degree of ***Bachelor of Engineering*** *in* ***Information Technology****, during the academic year* ***2024-2025*** under my guidance.

Dr. Pranjali P. Deshmukh
Guide
Information Technology Dept
PRMIT & R, Badnera.

Prof. (Dr) P. V. Ingole
HOD
Information Technology Dept
PRMIT & R, Badnera.

ABSTRACT

This report presents the end-to-end design and implementation of a custom programming language developed as part of the Compiler Design course. The primary goal of this project is to provide a hands-on learning experience in understanding how programming languages are created, from lexical analysis and parsing to semantic evaluation and execution. The language has been designed with an intuitive and beginner-friendly syntax, featuring unique keywords such as whynot for conditional branching, otherwise for alternative blocks, loop for iteration, and show for output operations. It supports essential constructs including variable declarations and assignments, arithmetic operations (including modulus), comparison operators, conditional statements, loop-based control flow, and string literals. Implemented using Java within the Eclipse IDE, the system consists of a custom tokenizer, parser, and interpreter. The interpreter processes user-written code stored in mycode files, enabling dynamic execution through a command-line utility (runMyCode filename.mycode). The parser uses regular expressions and recursive parsing techniques to validate syntax and enforce language rules, while the interpreter evaluates expressions and manages program state using a symbol table. This project not only deepens the understanding of compiler architecture but also demonstrates how high-level language features can be abstracted and implemented. The final result is a functional mini-language capable of executing meaningful programs with real logic, making it an effective and engaging educational tool. Future enhancements may include support for user-defined functions, arrays, type checking, and more advanced debugging and error reporting mechanisms.

Keywords: Custom Programming Language, Compiler Design, Java, Parser, Tokens, Syntax, If-Else, Loops, Print Statement, Simple Interpreter, Lexical Analysis

TABLE OF CONTENTS:

CHAPTER 1. INTRODUCTION

Programming languages are the foundation of all software systems. While we use high-level languages like Python, Java, or C++ every day, understanding how these languages are built provides deep insights into compiler construction, language syntax design, parsing techniques, and interpretation. This project was developed as part of the Compiler Design curriculum to explore and implement these core concepts by designing a custom, beginner-friendly programming language from scratch using Java.

The main motivation behind this project was to experience the inner workings of a compiler and interpreter firsthand, rather than just studying theoretical concepts. The custom language was designed to be easy to read and write, especially for educational use. It introduces new keywords and syntax that are intuitive, such as whynot for if, otherwise for else, loop for for, show for print, and mod for modulus operation. The language supports basic constructs like variable assignment, arithmetic expressions, conditional execution, and looping, making it powerful enough to solve small logic-based problems.

The project closely follows the compiler construction pipeline, including lexical analysis, syntax parsing, semantic evaluation, and execution, as outlined in foundational texts like [1]. The design of the custom language's syntax and semantics draws inspiration from practical compiler construction techniques, including the use of recursive parsing and symbol tables, as demonstrated in [4] and [6].

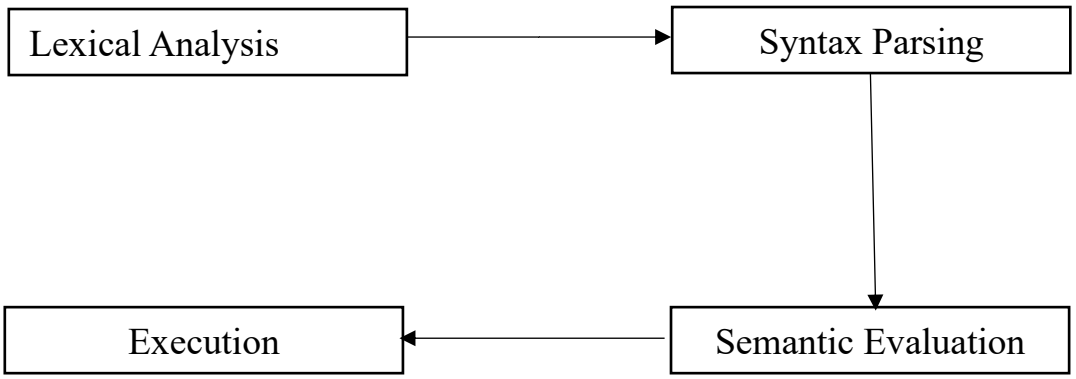


Fig. Working of a compiler

This introduction marks the beginning of a practical and highly informative journey through compiler design. It not only reinforces theoretical knowledge but also sharpens skills in language design, recursion, error handling, symbol tables, and more. The result is a working, self-defined mini programming language that can be used to demonstrate how actual compilers process and execute code.

- **Motivation**

  The primary motivation behind this project was to gain hands-on experience with compiler and interpreter internals in a fun, engaging, and educational way. Most programming languages use complex syntax and require large-scale compiler infrastructure. To simplify the learning process and make it beginner-friendly, this language was designed from scratch with intuitive, human-readable keywords such as whynot (if), otherwise (else), loop (for), and show (print). Creating this language not only deepened the understanding of compiler phases like lexical analysis, parsing, and semantic analysis but also strengthened problem-solving skills, especially in string processing and logical flow design.

- **Overview**

  This custom language allows users to perform variable assignments, arithmetic operations (including mod), conditional branching, iterative loops, and printing. Programs are written in .mycode files and executed using a command-line interpreter (runMyCode filename.mycode). Internally, the interpreter follows a structured pipeline that includes tokenization, syntax parsing using regular expressions, and execution using Java-based logic. Special care has been taken to maintain simplicity in both syntax and usage, making the language ideal for academic and educational use.

  Through this project, the complexities of compiler construction have been broken down into manageable components. It demonstrates how even a simple language can be structured, interpreted, and brought to life using foundational compiler principles. It also sets a path for future enhancements such as type checking, error reporting, and user-defined functions.

CHAPTER 2. OBJECTIVES

The main objective of this project is to design and implement a custom programming language that allows students to understand the internal workings of compilers and interpreters in a simplified and practical manner. By building this language from scratch, the project helps bridge the gap between theoretical compiler concepts and real-world application.

Key Objectives:

1. To design a beginner-friendly programming language with custom keywords and syntax suitable for educational purposes.
2. To understand and implement the core phases of compiler design, including lexical analysis, parsing, semantic validation, and interpretation.
3. To create a Java-based interpreter that can process .mycode files and execute the written code line-by-line.
4. To support essential programming constructs such as:
    - Variable assignment and arithmetic operations
    - Conditional statements using whynot and otherwise
    - Looping using loop ... from ... upto ... by ...:
    - Output display using the show keyword
5. To build a lightweight command-line tool (runMyCode filename.mycode) that mimics real-world interpreters and compilers.
6. To provide hands-on learning experience in compiler construction, symbol table management, syntax error handling, and control flow design.
7. To encourage further experimentation and innovation in language design, interpretation logic, and future compiler enhancements.

CHAPTER 3. TECHNOLOGIES USED

To design and implement the custom programming language, various technologies and tools were used to handle different aspects of the compiler and interpreter development process. Each of these played a specific and important role in making the language functional, readable, and executable.

- **Java:**

  Java was chosen as the primary programming language for implementing the compiler and interpreter due to its robust object-oriented features, powerful string manipulation capabilities, and platform independence. It was used to write the core components of the project such as the tokenizer (Lexer), parser, and the interpreter which processes and executes the .mycode files.
- **Eclipse IDE:**

  Eclipse served as the development environment where all Java files were created, compiled, and tested. It offers excellent features like syntax highlighting, real-time error detection, and debugging tools that helped speed up development and identify bugs early.
- **Command Line (Terminal):**

  The command line was used to run the program and execute .mycode files. A custom command format like runMyCode filename.mycode was developed to simulate the way real-world interpreters execute code, making the user interaction simple and effective.
- **Text Editors (Notepad, VS Code):**

  Basic text editors were used to write source code in the custom language using the .mycode extension. These editors made it easy to test various language features and write example programs.
- **Regular Expressions (Regex):**

  Regex played a key role in lexical analysis, helping to recognize patterns such as variable names, keywords (whynot, otherwise, loop, show, etc.), numbers, and operators. It was also used to validate the syntax of each line before parsing and execution.
- **Java File I/O (Input/Output):**

  Java's built-in file handling capabilities were used to read user-written .mycode files line by line. This allowed the interpreter to process the code sequentially and perform the required operations like variable assignments, looping, conditional checks, and printing.

Technology	Purpose
Java	Core programming language used to develop the parser, interpreter, and control logic.
Eclipse IDE	Integrated Development Environment used for writing, testing, and debugging Java code.
Command Line (Terminal)	Used to execute .mycode programs using a custom command runMyCode filename.mycode.
Text Editor	Any basic editor (e.g., Notepad, VS Code) can be used to write source code files in .mycode format.
Regular Expressions (Regex)	Used in lexical analysis to detect keywords, variables, numbers, and operators during tokenization.
File I/O in Java	Enables reading and processing the content of .mycode files line-by-line.

These technologies collectively provided a simple yet powerful environment for building, testing, and running the custom programming language. Java was chosen because of its strong string-handling capabilities and its object-oriented structure, which is ideal for building modular components like tokenizers and parsers.

CHAPTER 4. SYSTEM ARCHITECTURE

The modular design of the system aligns with modern compiler engineering practices described in [3]. The tokenizer's reliance on regular expressions for lexical analysis follows patterns from [1], while the parser's recursive descent approach aligns with [4]. The interpreter's real-time execution model mirrors principles from [5], and its modular structure reflects innovations like MLIR [8].

1. Source File (.mycode)

All programs are written in plain text files with a .mycode extension. These files contain code written using the custom syntax including variable assignments, loops, conditions, and print statements.

2. Input Handler (File Reader)

The system begins by loading the .mycode file using Java's file handling APIs. Each line of the file is read sequentially and passed on for further processing. This approach simulates how interpreters process code line-by-line instead of compiling the entire program at once.

3. Tokenizer (Lexical Analysis)

Once a line is read, it undergoes lexical analysis using **regular expressions**. This step breaks the line into tokens—small units such as keywords (whynot, loop, show), identifiers (variable names), numeric constants, operators (+, -, mod, etc.), and symbols (:=, ==, end;). The tokenizer ensures that the line only contains valid components before moving to syntax analysis.

4. Parser (Syntax Analyzer)

The parser takes the tokens and verifies whether the structure of the statement is grammatically correct according to the custom language rules. It identifies what type of statement it is—assignment, condition, loop, or output—and ensures it follows the defined syntax. For example:

- Variable assignment must follow: x := 5 + y
- Conditional block must start with: whynot x > 10: and end with end;
- Loops must follow: loop i from 1 upto 10 by 1:

If a syntax error is found, the parser immediately stops further processing and outputs an error message.

5. Interpreter (Executor)

Once the syntax is validated, the interpreter processes the statement:

- **For assignments**, it evaluates the expression and stores the result in a symbol table

(variable-value map).

- **For loops**, it repeatedly executes the inner block for the given range.
- **For conditions**, it evaluates the Boolean expression and executes the block only if the condition is true.
- **For show statements**, it prints the output to the console.

The interpreter handles the program flow just like any high-level scripting language.

6. Output Generator

If a line contains a show statement, the result is printed to the terminal. Additionally, any errors encountered during parsing or execution are displayed with appropriate messages for debugging.

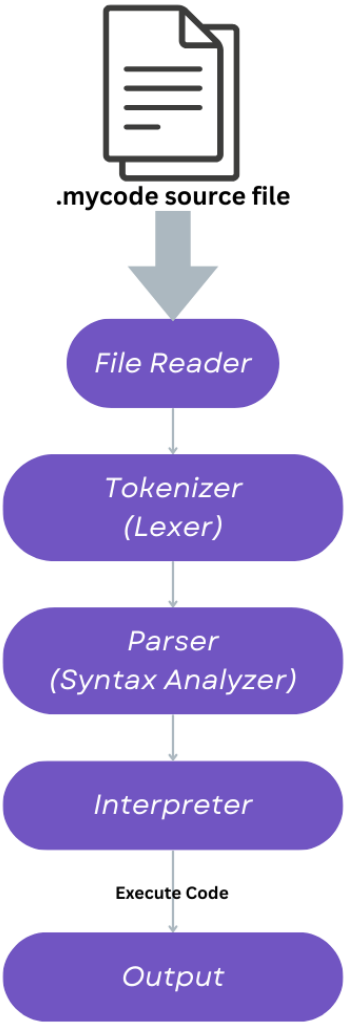


Fig. System Flow

This structured architecture allows you to simulate compiler principles in a simplified environment and encourages future expansions like error handling, data types, or even function support.

CHAPTER 6. IMPLEMENTATION DETAILS

The custom programming language was implemented in Java, with its core functionality split across four major components: the Main program, the Lexer, the Parser, and the Interpreter. Each component is responsible for a different stage in the code execution pipeline, closely following the structure of a typical interpreter.

1. Main Program (Main.java)

This serves as the entry point for the language. It allows the user to run a command like:

```bash
runMyCode filename.mycode
```

The main class handles file input, reads the content of the .mycode file, passes it to the lexer for tokenization, and then to the parser for syntax validation and execution. It includes basic input validation and exception handling for missing files or syntax errors.

2. Lexer (Lexer.java)

The Lexer (or tokenizer) is responsible for scanning the raw source code and converting it into a list of tokens. It uses regular expressions to match keywords (whynot, loop, show, end;, etc.), operators (+, -, mod, etc.), identifiers (variable names), string literals, numbers, and comparison operators.

Example token types:

- PRINT, IF, ELSE, FOR, MOD
- NUMBER, STRING, IDENTIFIER, ASSIGN, OPERATOR, COMPARISON, PAREN, SEMICOLON

The lexer outputs a structured list of tokens, each with a type and value, which the parser then consumes.

3. Parser (Parser.java)

The parser is the core of your language's logic and is responsible for:

- Validating syntax of each statement.
- Executing logic inline (you've combined parsing and interpretation).
- Managing variable storage using a symbol table (Map<String, String>).

- Evaluating expressions and conditions.
- Handling control structures:
    - whynot ... otherwise ... end; (if-else)
    - loop i from X upto Y by Z: (for loop)
- Supporting string assignments and printing using the show statement.

The parser has modular functions like:

- parseAssignment() for handling :=
- parsePrint() for handling show(...)
- parseIfElse() for conditional blocks
- parseLoop() for loop blocks
- evaluateArithmeticExpression() for computing values
- evaluateComparison() for conditions like x > y

All execution happens immediately after parsing — this design makes your parser a recursive descent interpreter.

4. Interpreter (Interpreter.java)

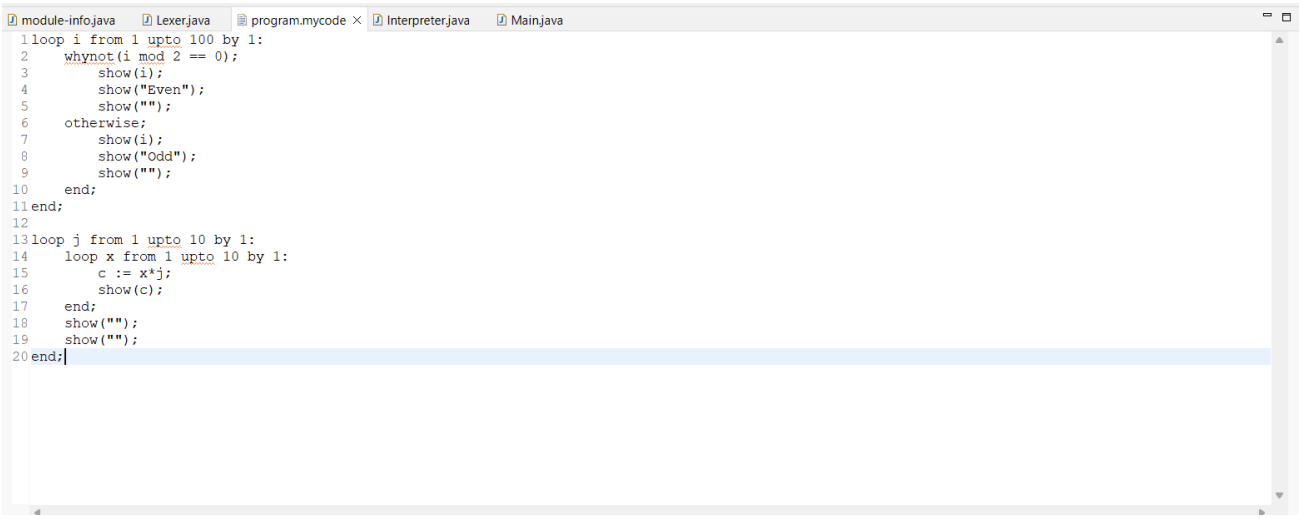
Although some of the interpretation logic was moved to the Parser, your Interpreter.java acts as a simpler interface for executing basic assignments and print statements based on token input. It maintains a Map<String, Integer> to store variable values and can evaluate basic assignment and print statements.

The combined parser-interpreter design reflects optimizations discussed in [5] (SSA-based techniques) and [8] (MLIR's infrastructure). Error handling and symbol table management were influenced by [4] and [6].

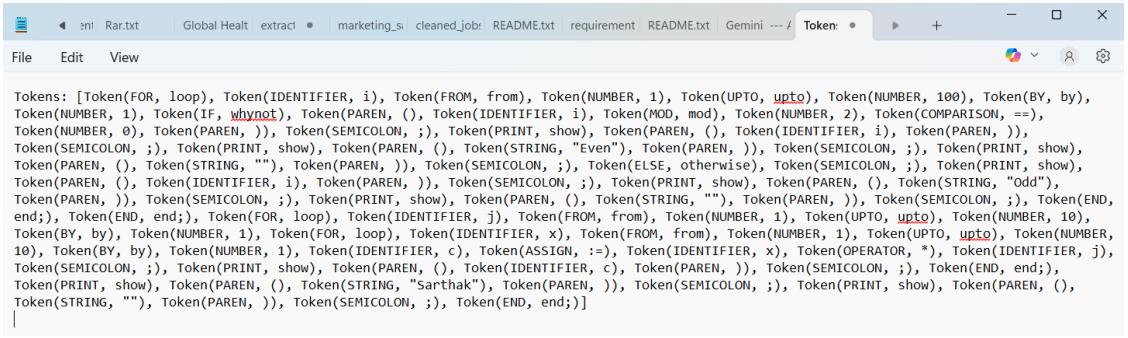
CHAPTER 7. OUTPUT

Executing the code:

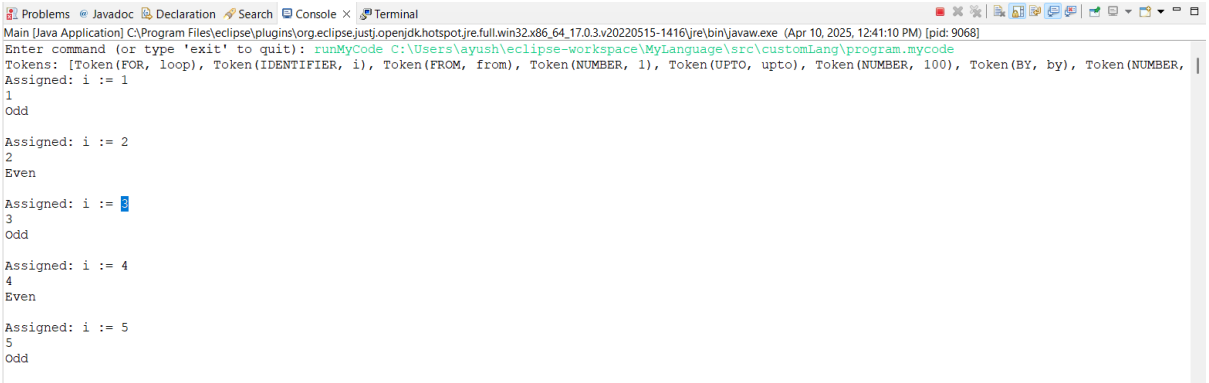
Input file:



Tokenization:



Final Output after parsing and executing the .mycode file :



CONCLUSION

The development of this custom programming language has been a highly educational and rewarding experience. It provided a practical understanding of the core concepts in compiler design, such as lexical analysis, parsing, and interpretation. By building the language from the ground up using Java, the project not only reinforced theoretical concepts but also enhanced problem-solving and software design skills.

The language successfully supports essential programming constructs including variable assignments, arithmetic expressions, conditional branching, loops, and print statements—all structured using a unique and beginner-friendly syntax. The use of custom keywords like whynot, otherwise, loop, and show made the language intuitive and enjoyable to work with, especially for learners.

Implementing the interpreter logic separately and integrating it with the parser improved code organization and modularity. The project also highlighted the importance of error handling, expression evaluation, and symbol table management in interpreter-based languages.

Overall, this project serves as a strong foundation for further exploration into advanced compiler topics such as abstract syntax trees (ASTs), optimization techniques, type checking, and code generation. Future enhancements could leverage frameworks like MLIR [8] or SSA-based optimizations [5]. The project's educational value aligns with the goals highlighted in [6].

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Pearson Education, 2006.

[2] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[3] K. D. Cooper and L. Torczon, *Engineering a Compiler*, 2nd ed. Elsevier, 2011.

[4] A. I. Holub, *Compiler Design in C*. Prentice Hall, 1990.

[5] F. Rastello and F. Bouchez, *SSA-Based Compiler Design*. Springer, 2022.

[6] B. S. Abubakar, "An Overview of Compiler Construction," ResearchGate, 2021. [Online]. Available: https://www.researchgate.net/publication/...

[7] M. B. Dhore and S. S. Sonavane, "Advancements in Compiler Design and Optimization Techniques," *E3S Web of Conferences*, vol. X, no. Y, p. ZZZZ, 2023. [Online]. Available: https://www.e3s-conferences.org/...

[8] C. Lattner et al., "MLIR: A Compiler Infrastructure for the End of Moore's Law," *arXiv*, 2020. [Online]. Available: https://arxiv.org/abs/2002.11054

[9] Oracle, *Java Platform, Standard Edition Documentation*. Oracle Corporation, 2024. [Online]. Available: https://docs.oracle.com/javase/specs/