Building a Tree-Walking Interpreter for Lox: Design Document

Overview

This document outlines the architecture for a complete interpreter for the Lox programming language, implementing a tree-walking evaluation strategy. The key architectural challenge is designing a clean separation between the static structure of the code (lexing, parsing) and its dynamic execution (environment-based evaluation), while managing state, scope, and first-class functions in a way that is both correct and educational for the implementer.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

> **Milestone(s):** All milestones (foundational context)

1. Context and Problem Statement

Interpreting a programming language is the art of imbuing inert text with life. The fundamental problem is transforming a linear sequence of characters—a program—into a dynamic, stateful computation that produces effects (like printing to a screen, modifying a file, or controlling a robot). This transformation is not direct; it requires bridging a profound conceptual gap between the **static syntax** of the language (its grammar and keywords) and the **dynamic runtime** (the execution of operations on data in memory). This section establishes the core architectural philosophy for our Lox interpreter: the tree-walking strategy, a deliberate design that prioritizes clarity and pedagogical value while solving this fundamental problem.

Mental Model: The Tour Guide

Imagine you are handed a detailed, hierarchical map of a complex city (**the Abstract Syntax Tree, or AST**). The map doesn't *do* anything; it's just a static representation of the city's layout—streets (expressions), landmarks (literals), and districts (statements). A **tree-walking interpreter** acts as your personal tour guide. The guide walks you through this map step-by-step, starting at the main entrance (the program entry point). At each landmark on the map, the guide consults a comprehensive rulebook (**the language semantics**) that dictates what action to perform at that specific type of location.

For example, upon reaching a `+` intersection (a Binary expression node), the rulebook states: "First, walk down the left street and report back what you find. Then, walk down the right street and report back. Finally, if both reports are numbers, add them; if both are strings, join them." The guide faithfully executes these instructions, carrying a notebook (**the Environment**) to remember variable values assigned at different addresses. This direct, recursive walk through the map's structure is intuitive—the execution flow mirrors the code's syntactic structure. However, it can be inefficient, as the guide might re-walk the same paths (like in a loop) without remembering the terrain.

The Core Challenge: Bridging Syntax and Runtime

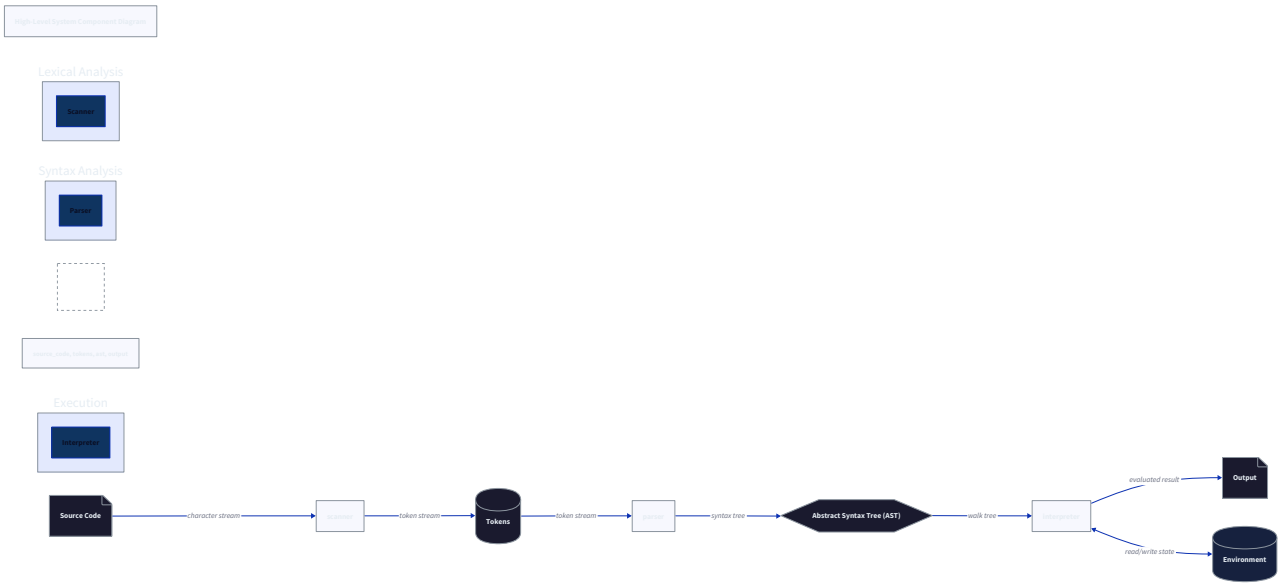
Source code is a one-dimensional string. Runtime behavior is multidimensional, involving control flow jumps, mutable state, and nested function calls. The core architectural challenge is designing a pipeline that systematically translates the former into the latter. A single-step translation is impossible for all but the most trivial languages due to complexities like nested expressions, scoped variables, and forward references.

Our solution is a **multi-phase design**, decomposing the problem into three sequential transformations, each building on the output of the previous phase. This separation of concerns is a classic pattern in language implementation:

1. **Lexical Analysis (Scanner):** The first transformation converts the raw character stream into a stream of meaningful **tokens**. This phase strips away whitespace and comments, and recognizes the basic vocabulary of the language: keywords (`if` , `while`), identifiers (`count`), literals (`42` , `"hello"`), and operators (`+` , `==`). It's akin to breaking a sentence into individual words and punctuation.
2. **Syntactic Analysis (Parser):** The second transformation organizes the flat token stream into a hierarchical **Abstract Syntax Tree (AST)**. This phase enforces the grammar rules of the language, ensuring tokens appear in a valid order and establishing the nesting relationships dictated by parentheses and operator precedence. It's like diagramming a sentence to identify subject, verb, object, and subordinate clauses.

3. **Semantic Analysis & Execution (Interpreter):** The final transformation walks the AST and performs the actual computations and side effects. This phase resolves the *meaning* of the syntactic constructs: evaluating expressions, executing statements, managing variable storage in **Environments**, and directing control flow. It's the tour guide bringing the map to life.

The data flows linearly through this pipeline: `Source Code (String)` → `Scanner` → `List<Token>` → `Parser` → `List<Stmt>` → `Interpreter` → `Runtime Effects/Output`.



Why is this separation necessary? Each phase operates at a different level of abstraction and has a distinct, testable responsibility. The scanner worries about character-level patterns but not grammar. The parser worries about grammar but not what `+` *does*. The interpreter worries about semantics and state but not about where parentheses go. This modularity makes the system easier to understand, debug, and extend. For instance, adding a new operator like `**` requires: 1) teaching the scanner to recognize `**` as a token, 2) teaching the parser its precedence level, and 3) teaching the interpreter its mathematical meaning—all in isolated components.

Existing Interpreter Architectures

The tree-walking interpreter is one of several architectures for implementing a programming language. The choice among them represents a classic engineering trade-off between simplicity, performance, and flexibility.

Decision: Choosing a Tree-Walking Interpreter Architecture

- **Context**: We are building an interpreter for Lox primarily as an educational project. The primary goal is to understand language implementation concepts from the ground up, not to achieve peak execution speed. We need an architecture that maps clearly to the language's syntax and semantics, making the connection between source code and runtime behavior as transparent as possible.
- **Options Considered**:
  1. **Tree-Walking Interpreter**: Directly traverses the AST, executing operations at each node.
  2. **Bytecode Virtual Machine (VM)**: Compiles the AST to a compact, linear bytecode instruction set, which is then executed by a virtual stack- or register-based machine.
  3. **Ahead-of-Time (AOT) or Just-in-Time (JIT) Compiler**: Translates source code directly to native machine code for the host CPU (e.g., x86, ARM).
- **Decision**: We will implement a **Tree-Walking Interpreter**.
- **Rationale**:
  - **Conceptual Clarity**: The execution path of a tree-walker directly mirrors the syntactic structure of the source code. A `while` loop node in the AST leads directly to a loop in the interpreter's execution. This 1:1 mapping is invaluable for learning.
  - **Implementation Simplicity**: It requires fewer moving parts. There is no need to design a bytecode instruction set, a compiler from AST to bytecode, or a virtual machine with its own instruction dispatch loop. The interpreter is essentially a large recursive function over the AST.
  - **Incremental Development**: Features can be added by extending the AST and adding a corresponding `visit` method in the interpreter. This aligns perfectly with the milestone-based learning approach.
- **Consequences**:
  - **Performance**: Tree-walking is slower than a bytecode VM. Each AST node involves a virtual method dispatch (via the Visitor pattern), and complex expressions require traversing a deep tree for every evaluation (e.g., in a tight loop).
  - **No Intermediate Representation**: The lack of a bytecode intermediate representation (IR) makes some optimizations (like constant folding or basic block analysis) more awkward to implement, though they are less of a priority for this project.
  - **Direct Semantics**: The interpreter's logic is the "specification" of the language. This can be both a pro (easy to reason about) and a con (runtime errors are deeply embedded in the traversal logic).

The following table compares the key architectural approaches:

Architecture	How it Works	Pros	Cons	Best For
Tree-Walking Interpreter	Recursively evaluates nodes in the Abstract Syntax Tree (AST).	**Simple to implement and understand.** Direct mapping from syntax to execution. Easy to add new language features.	**Slow.** High overhead per operation due to tree traversal and visitor pattern dispatch. No easy path to optimization.	**Educational projects, prototyping, scripting languages** where simplicity and clarity are paramount over raw speed.
Bytecode Virtual Machine	Compiles AST to a dense, linear bytecode. A virtual CPU (stack-based or register-based) executes the bytecode.	**Much faster than tree-walking.** Bytecode is compact, and the dispatch loop is efficient. Enables optimizations at the bytecode level. Foundation for JIT compilation.	**More complex.** Requires designing a bytecode ISA, a compiler, and a VM. The mapping from source to execution is less direct.	**Production interpreters** (e.g., CPython, Lua, the Java JVM). Balances performance with portability.
Ahead-of-Time (AOT) Compiler	Translates source code directly to native machine code for a specific CPU architecture.	**Maximum runtime performance.** Executes directly on hardware, no interpreter loop overhead.	**Extremely complex.** Must handle low-level details of the target CPU (registers, instruction selection). Loss of portability; output is platform-specific. Long compilation time.	**Systems programming languages** (C, C++, Rust) and performance-critical applications.
Just-in-Time (JIT) Compiler	Starts as an interpreter or bytecode VM, but profiles "hot" code paths and dynamically compiles them to native code during execution.	**Can approach AOT performance** for hot code while maintaining the flexibility of an interpreter for cold code. Adaptive optimization based on runtime profiling.	**Extreme complexity.** Combines challenges of VM and compiler design. High memory usage for storing both bytecode and native code.	**High-performance managed runtimes** (JavaScript V8, Java HotSpot, .NET CLR).

Our chosen path, the tree-walking interpreter, is the **simplest gateway** into the world of language implementation. It allows us to focus on the core concepts—lexing, parsing, environments, closures, and classes—without the additional cognitive load of designing a bytecode format and a virtual machine. The performance trade-off is acceptable for an educational interpreter; Lox programs will be plenty fast for learning and scripting purposes. Furthermore, the skills and patterns learned here (especially the Visitor pattern for AST traversal and environment management for scope) are directly transferable to building more advanced bytecode VMs or compilers in the future.

Implementation Guidance

This section is foundational and does not involve writing code. However, establishing a solid project structure from the outset is critical. Below is the recommended Java package and directory layout that aligns with the high-level architecture. We will flesh out these directories in subsequent component design sections.

Recommended File/Module Structure:

```
lox/
├── src/main/java/com/craftinginterpreters/lox/
│   ├── Lox.java                        # Main entry point, coordinates scanning, parsing, interpreting
│   │
│   ├── scanner/                        # Milestone 1: Scanner (Lexer)
│   │   ├── Scanner.java
│   │   ├── Token.java
│   │   └── TokenType.java
│   │
│   ├── parser/                         # Milestones 2 & 3: Parser & AST
│   │   ├── Parser.java
│   │   └── ParseError.java
│   │
│   ├── ast/                            # Milestone 2: Abstract Syntax Tree nodes
│   │   ├── Expr.java                   # Base expression class
│   │   ├── Stmt.java                   # Base statement class
│   │   ├── expr/                       # Concrete expression subclasses
│   │   │   ├── Assign.java
│   │   │   ├── Binary.java
│   │   │   ├── Call.java
│   │   │   ├── Get.java
│   │   │   ├── Grouping.java
│   │   │   ├── Literal.java
│   │   │   ├── Logical.java
│   │   │   ├── Set.java
│   │   │   ├── Super.java
│   │   │   ├── This.java
│   │   │   ├── Unary.java
│   │   │   └── Variable.java
│   │   └── stmt/                       # Concrete statement subclasses
│   │       ├── Block.java
│   │       ├── Class.java
│   │       ├── Expression.java
│   │       ├── Function.java
│   │       ├── If.java
│   │       ├── Print.java
│   │       ├── Return.java
│   │       ├── Var.java
│   │       └── While.java
│   │
│   ├── interpreter/                    # Milestones 4-10: Interpreter & Runtime
│   │   ├── Interpreter.java            # Main tree-walking visitor
│   │   ├── RuntimeError.java
│   │   ├── environment/                # Milestone 5: Environments
│   │   │   └── Environment.java
│   │   └── runtime/                    # Runtime value representations
│   │       ├── LoxCallable.java        # Interface for functions/classes
│   │       ├── LoxClass.java
│   │       ├── LoxFunction.java
│   │       ├── LoxInstance.java
│   │       └── Return.java             # Control flow for return statements
│   │
│   └── tools/                          # Utilities
│       └── AstPrinter.java             # Milestone 2: Pretty-printer for debugging
│
└── src/test/java/com/craftinginterpreters/lox/   # Unit tests for each component
    ├── scanner/
    ├── parser/
    └── interpreter/
```

Language-Specific Hints (Java):

- Use `enum` for `TokenType` to represent the finite set of token categories (keywords, operators, etc.).
- The Visitor pattern for AST traversal is a natural fit in Java using abstract classes and accept/visit methods. While it involves some boilerplate, it provides type safety and clean separation between the AST structure and operations on it.
- For representing runtime values, consider using `Object` as the base type in the interpreter, with explicit casting and `instanceof` checks. Alternatively, you can define a base `LoxValue` interface. We will explore this decision in the Data Model section.

- Use `java.util.HashMap<String, Object>` for the core storage within an `Environment`.
- **Milestone Checkpoint**: After setting up this structure, you should be able to compile the project. A simple test is to create a `Lox.java` file with a `main` method that prints a message. Run `javac` on the source directory and execute the main class to verify your build environment is configured.

> **Milestone(s):** All milestones (foundational goals)

2. Goals and Non-Goals

This section defines the precise boundaries of the interpreter project, establishing what constitutes a successful implementation. By explicitly stating functional requirements, non-functional priorities, and out-of-scope features, we create a clear target for development and prevent scope creep in this educational endeavor. The goals are structured to align with the 10 milestones, which incrementally build from lexical analysis to inheritance.

Functional Goals

The interpreter must fully implement the Lox programming language as defined in *Crafting Interpreters*. Lox is a dynamically-typed, object-oriented scripting language designed to be small enough for a single implementer yet complete enough to illustrate core language implementation concepts. The functional requirements are broken down by language feature category, each corresponding to a specific milestone.

> **Mental Model: The Language Specification Checklist** Think of the Lox language specification as a checklist of features that a "complete" Lox interpreter must support. Our implementation is a faithful translation of that checklist into executable code, feature by feature, milestone by milestone. Each checked item moves us closer to a fully functional language.

The following table enumerates the mandatory language features, their corresponding implementation milestones, and a precise description of what "support" entails:

Feature Category	Milestone	Implementation Requirement Description
Lexical Structure	1 (Scanner)	The scanner must tokenize all Lox lexical elements: keywords (`var` , `fun` , `if` , `else` , `while` , `for` , `return` , `class` , `super` , `this` , `and` , `or` , `print` , `nil` , `true` , `false`), identifiers, literals (strings with escape sequences, numbers with decimal points), operators (`+` , `-` , `*` , `/` , `!` , `=` , `==` , `!=` , `<` , `<=` , `>` , `>=`), and punctuation (`(` , `)` , `{` , `}` , `,` , `.` , `;`). Whitespace and comments (both `//` line and `/* */` block) must be ignored.
Expressions & Precedence	3 (Parsing Expressions)	The parser must correctly parse all expression types with proper operator precedence and associativity. Precedence levels (from lowest to highest) are: assignment (`=`), logical (`or` , `and`), equality (`==` , `!=`), comparison (`<` , `<=` , `>` , `>=`), term (`+` , `-`), factor (`*` , `/`), and unary (`!` , `-`). Parentheses must override precedence.
Basic Evaluation	4 (Evaluating Expressions)	The interpreter must evaluate expressions to produce runtime values: arithmetic operations on numbers, string concatenation with `+` , unary negation and logical NOT, truthiness rules (`nil` and `false` are falsy, everything else truthy), and comparison/equality operations with appropriate type checking.
Variables & State	5 (Statements and State)	Support variable declaration (`var x = value;`), assignment (`x = value;`), and scoping via nested environments. The `print` statement must output values to standard output. Expression statements must evaluate expressions for side effects.
Control Flow	6 (Control Flow)	Implement `if` / `else` conditional execution, `while` loops, `for` loops (desugared to `while`), and logical operators `and` / `or` with short-circuit evaluation.
First-Class Functions	7 (Functions)	Support function declaration (`fun name(parameters) { body }`), function calls, `return` statements, and treating functions as first-class values (assignable to variables, passable as arguments, returnable from functions). Recursion must work.
Lexical Closures	8 (Closures)	Functions must capture their **lexical environment** at definition time, creating closures that retain access to variables from outer scopes even after those scopes have exited. Nested functions must be supported.
Classes & Instances	9 (Classes)	Support class declarations (`class Name { methods }`), instance creation (`ClassName()`), property access (`instance.field`), method calls (`instance.method()`), and the `this` keyword bound to the instance within methods. An `init()` method serves as the constructor.
Single Inheritance	10 (Inheritance)	Support subclassing via `class Derived < Base` . Derived classes inherit methods from their superclass. The `super` keyword must invoke superclass methods from within overridden methods. `super.init()` must be callable within a derived class's `init` .

These nine categories represent the complete syntactic and semantic feature set of the Lox language. A successful interpreter must correctly execute Lox programs utilizing any combination of these features.

Non-Functional Goals & Non-Goals

While functional goals define *what* the interpreter does, non-functional goals define *how well* it does them and what qualities we prioritize. For an educational project, clarity, simplicity, and learning value take precedence over performance, robustness, or production readiness.

Non-Functional Goals (Priorities)

1. **Clarity and Readability of Implementation:** The code should be structured to be understandable by a developer learning interpreter construction. This means favoring explicit, well-named functions over clever optimizations, using consistent patterns (like the Visitor pattern for AST traversal), and adding explanatory comments for non-obvious algorithms.
2. **Educational Value:** Each component should illustrate a fundamental concept of language implementation (lexical analysis, recursive descent parsing, environment-based evaluation, closure capture, etc.). The architecture should make these concepts visible and decoupled.
3. **Incremental Buildability:** The design must support implementation in the order of the 10 milestones, where each milestone yields a testable, functional subset of the interpreter. Earlier milestones should not require knowledge or infrastructure from later ones.
4. **Helpful Error Reporting:** When the interpreter encounters an error (lexical, syntactic, or runtime), it should report the error with a clear message and the precise source location (line number, ideally column). This aids debugging for both the interpreter user and the

implementer.

5. **Correctness for the Lox Specification:** The interpreter's behavior must match the semantics described in *Crafting Interpreters*. Edge cases (e.g., truthiness, string concatenation with non-strings, inheritance chains) should be handled as specified.

Non-Goals (Explicitly Out of Scope)

The following are explicitly **not** goals for this project. Including them would expand scope beyond the core educational objectives, add significant complexity, and distract from understanding the foundational concepts.

Non-Goal Category	Specific Exclusions	Rationale
Performance	No performance optimization (e.g., bytecode compilation, JIT, AST caching, hash table optimization), no benchmarking against other interpreters.	The primary aim is education, not speed. A tree-walking interpreter is inherently slower than a bytecode VM; optimizing it would add complexity without illuminating new core concepts.
Tooling & Ergonomics	No REPL (Read-Eval-Print Loop), no debugger, no syntax highlighting, no language server protocol support.	These are valuable tools but are separate projects that build *upon* a working interpreter. They would significantly increase scope.
Standard Library	No built-in functions (e.g., `clock()`, `input()`, `type()`), no data structures beyond classes, no I/O beyond `print`.	Lox as defined in the book has a minimal runtime. Adding a standard library involves designing APIs and implementing native functions, which is a separate topic (native interop).
Memory Management	No garbage collection implementation. We rely on the host language's (e.g., Java) garbage collector to reclaim unused `LoxInstance`, `LoxFunction`, and `Environment` objects.	Implementing a garbage collector is a major project unto itself. For simplicity, we assume automatic memory management by the host platform.
Native Extensions	No ability to call functions written in the host language (Java/C/Rust) from Lox, nor to embed the interpreter in a larger application.	Interoperability introduces complex binding and type conversion logic. The project focuses on implementing a self-contained language.
Language Extensions	No additional syntax or features beyond standard Lox (e.g., no `+=` operators, no `switch` statements, no `try`/`catch`, no modules).	Staying true to the book's specification ensures the project remains focused and comparable to reference implementations.
Concurrency	No threads, no async/await, no parallelism.	Concurrency introduces immense complexity in state management and is orthogonal to the core interpreter pipeline.
Robust Production Deployment	No sandboxing, no security model, no support for multi-file programs/imports, no detailed logging or monitoring.	This is a learning project, not a production engine.

ADR: Tree-Walking Interpreter vs. Bytecode Virtual Machine

- **Context:** We must choose the core execution strategy for our Lox interpreter. The two primary patterns presented in *Crafting Interpreters* are the tree-walking interpreter (Part I) and the bytecode virtual machine (Part II).
- **Options Considered:**
  1. **Tree-Walking Interpreter:** Directly traverses the AST, executing nodes by recursively evaluating their children and combining results.
  2. **Bytecode Virtual Machine:** Compiles the AST to a dense bytecode instruction stream, then executes it using a stack-based virtual machine.
- **Decision:** Implement a **tree-walking interpreter**.
- **Rationale:** The tree-walking approach has a more direct correspondence between the source code structure (AST) and execution logic, making it easier to understand, debug, and incrementally build. It requires less upfront infrastructure (no compiler, no bytecode definition, no VM loop). It perfectly serves the educational goal of understanding semantic evaluation without the added complexity of an intermediate representation and virtual machine.
- **Consequences:** The interpreter will be simpler to implement and understand but will be slower (often 5-10x) than a bytecode VM. This is an acceptable trade-off for learning. The design cleanly separates scanning, parsing, and interpreting phases.

Option	Pros	Cons	Chosen?
Tree-Walking Interpreter	Direct mapping from AST to execution; simpler to implement and debug; less code; ideal for incremental milestones.	Slower execution speed; repeated traversal of AST nodes; less illustrative of production interpreter techniques.	**Yes**
Bytecode Virtual Machine	Faster execution; introduces important concepts of compilation and VM design; more realistic for production languages.	Significantly more complex; requires designing bytecode, compiler, and VM; harder to debug; larger upfront investment.	No

By adhering to these goals and non-goals, the project remains a focused, educational journey through the essential components of a dynamic language interpreter, providing maximum learning value within a manageable scope.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option (Recommended)	Advanced Option (If Curious)
Project Structure	Single-language monolithic project (Java).	Multi-language comparison (implement scanner in C, parser in Java, interpreter in Rust).
Build System	Maven or Gradle for Java; simple `Makefile` for C; Cargo for Rust.	Custom build scripts integrating multiple languages.
Testing Framework	JUnit (Java), Catch2 (C), `cargo test` (Rust). Use simple assertion libraries.	Property-based testing (e.g., jqwik for Java) for generative tests.
Error Reporting	Print errors with line numbers to `stderr`.	Fancy error formatting with source code snippets and underlines.

B. Recommended File/Module Structure: Organize your code by the logical phases of the interpreter. This separation of concerns makes the codebase navigable and mirrors the architectural diagram.

```
lox-interpreter/                  # Project root
├── src/main/java/com/craftinginterpreters/lox/
│   ├── Lox.java                  # Main entry point: reads file, drives pipeline
│   ├── Token.java                # Token data class (type, lexeme, literal, line)
│   ├── TokenType.java            # Enum of all token types (IF, PLUS, IDENTIFIER, etc.)
│   │
│   ├── scanner/                  # Milestone 1
│   │   └── Scanner.java          # Converts source string to List<Token>
│   │
│   ├── ast/                      # Milestone 2
│   │   ├── Expr.java             # Abstract base class for expressions
│   │   ├── Stmt.java             # Abstract base class for statements
│   │   ├── *.java                # Concrete subclasses: Binary, Unary, Literal, Var, etc.
│   │   └── AstPrinter.java       # Visitor that prints AST as S-expressions
│   │
│   ├── parser/                   # Milestone 3
│   │   └── Parser.java           # Recursive descent parser: List<Token> -> List<Stmt>
│   │
│   ├── interpreter/              # Milestones 4-10
│   │   ├── Interpreter.java      # Main tree-walking interpreter, implements Expr.Visitor<Void>, Stmt.Visitor<Void>
│   │   ├── Environment.java      # Chain of scopes for variable storage
│   │   ├── RuntimeError.java     # Exception for runtime errors (e.g., type mismatch)
│   │   ├── LoxCallable.java      # Interface for functions/classes
│   │   ├── LoxFunction.java      # Represents a user-defined function (closure)
│   │   ├── LoxClass.java         # Represents a class
│   │   └── LoxInstance.java      # Represents an instance of a class
│   │
│   └── utils/                    # Utilities
│       └── ErrorReporter.java    # Centralized error reporting (optional)
├── lib/                          # Dependencies (if any)
└── test/                         # Unit tests (parallel to src structure)
```

C. Infrastructure Starter Code (COMPLETE, ready to use): Here is a complete, minimal `Token` class and `TokenType` enum as they are foundational and not the core learning challenge of any milestone.

JAVA

```java
// TokenType.java

package com.craftinginterpreters.lox;


public enum TokenType {

    // Single-character tokens.

    LEFT_PAREN, RIGHT_PAREN, LEFT_BRACE, RIGHT_BRACE,

    COMMA, DOT, MINUS, PLUS, SEMICOLON, SLASH, STAR,


    // One or two character tokens.

    BANG, BANG_EQUAL,

    EQUAL, EQUAL_EQUAL,

    GREATER, GREATER_EQUAL,

    LESS, LESS_EQUAL,


    // Literals.

    IDENTIFIER, STRING, NUMBER,


    // Keywords.

    AND, CLASS, ELSE, FALSE, FUN, FOR, IF, NIL, OR,

    PRINT, RETURN, SUPER, THIS, TRUE, VAR, WHILE,


    EOF

}
```

```java
// Token.java

package com.craftinginterpreters.lox;


public class Token {

    public final TokenType type;

    public final String lexeme;

    public final Object literal;

    public final int line;


    public Token(TokenType type, String lexeme, Object literal, int line) {

        this.type = type;

        this.lexeme = lexeme;

        this.literal = literal;

        this.line = line;

    }


    public String toString() {

        return type + " " + lexeme + " " + literal;

    }

}
```

D. Core Logic Skeleton Code (for the main driver): The `Lox` class is the entry point. It demonstrates the high-level pipeline.

JAVA

```java
// Lox.java

package com.craftinginterpreters.lox;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.nio.charset.Charset;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.List;

public class Lox {

    // TODO 1: Add a static errorReporter field or use a global flag to track if an error occurred.

    static boolean hadError = false;

    static boolean hadRuntimeError = false;

    public static void main(String[] args) throws IOException {

        if (args.length > 1) {

            System.out.println("Usage: jlox [script]");

            System.exit(64);

        } else if (args.length == 1) {

            runFile(args[0]);

        } else {

            runPrompt();

        }

    }

    private static void runFile(String path) throws IOException {

        byte[] bytes = Files.readAllBytes(Paths.get(path));

        run(new String(bytes, Charset.defaultCharset()));

        // TODO 2: If there was a scan or parse error, exit with code 65 (data error).

        if (hadError) System.exit(65);

        // TODO 3: If there was a runtime error, exit with code 70 (software error).

        if (hadRuntimeError) System.exit(70);

    }

    private static void runPrompt() throws IOException {

        InputStreamReader input = new InputStreamReader(System.in);
```

```
        BufferedReader reader = new BufferedReader(input);

        for (;;) {
            System.out.print("> ");
            String line = reader.readLine();
            if (line == null) break;
            run(line);
            // TODO 4: Reset error flag for interactive session so errors don't kill the REPL.
            hadError = false;
        }
    }

    private static void run(String source) {
        // TODO 5: Instantiate the Scanner with the source string.
        // Scanner scanner = new Scanner(source);
        // TODO 6: Call scanner.scanTokens() to get a List<Token>.
        // List<Token> tokens = scanner.scanTokens();

        // TODO 7: Instantiate the Parser with the tokens.
        // Parser parser = new Parser(tokens);
        // TODO 8: Call parser.parse() to get a List<Stmt>.
        // List<Stmt> statements = parser.parse();

        // TODO 9: Stop if there was a parse error (indicated by hadError).
        // if (hadError) return;

        // TODO 10: Instantiate the Interpreter.
        // Interpreter interpreter = new Interpreter();
        // TODO 11: Call interpreter.interpret(statements) to execute the program.
        // interpreter.interpret(statements);
    }

    // TODO 12: Implement error reporting methods (static) that print to stderr and set hadError.
    // public static void error(int line, String message) { ... }
    // public static void error(Token token, String message) { ... }
    // public static void runtimeError(RuntimeError error) { ... }
}
```

E. Language-Specific Hints (Java):

- Use `java.util.HashMap` for `Environment` values map and `LoxInstance` fields.
- Use `java.util.List` for sequences (tokens, statements, parameters).
- For the Visitor pattern, define interfaces `Expr.Visitor<R>` and `Stmt.Visitor<R>` with visit methods for each node type. Have `Interpreter` implement these interfaces.
- Use `Double` for Lox number values. Be careful with equality comparisons (`==` on `Double` objects); consider checking for `double` values within an epsilon for numeric equality, but for simplicity, you can use `equals` for Lox's `==` operator (but note `Double.NaN`).
- Represent Lox `nil` as Java `null` . Represent Lox booleans as Java `Boolean` .
- For runtime errors, define a custom `RuntimeError` exception class that extends `RuntimeException` and holds a `Token` for location.

F. Milestone Checkpoint (Verification for Section 2): After setting up the project structure and the basic `Lox` driver with the skeleton above, you can verify your environment works:

- **Command:** `javac -d out src/main/java/com/craftinginterpreters/lox/*.java src/main/java/com/craftinginterpreters/lox/scanner/*.java ...` (or use your build tool)
- **Expected:** Successful compilation with no errors (just warnings about unused imports/todos).
- **Manual Test:** Create a simple Lox file `test.lox` with `print "Hello, world!";` . Run your interpreter (once you implement the pipeline). You should see `"Hello, world!"` printed.
- **Signs of Trouble:** If you encounter `ClassNotFoundException` or `NoClassDefFoundError` , check your classpath and package declarations.

G. Debugging Tips for Goals Alignment:

Symptom	Likely Cause	How to Diagnose	Fix
Interpreter supports `+=` operator.	Implemented language extension beyond Lox spec.	Review scanner and parser for non-standard tokens/grammar rules.	Remove support for `+=` to stay true to the book's Lox language.
Function does not capture outer variables (closure broken).	Environment capture strategy is incorrect (copy vs. reference).	Print environment chain during function call; check if parent environment is the defining one.	Ensure `LoxFunction` stores a reference to the *defining* environment, not a copy or the current one.
Interpreter is extremely slow on loops.	Tree-walking interpreter overhead.	Profile to confirm; this is expected.	Acceptable for learning. If performance is critical, consider later moving to a bytecode VM (non-goal for this project).

> **Milestone(s):** All milestones (foundational architecture)

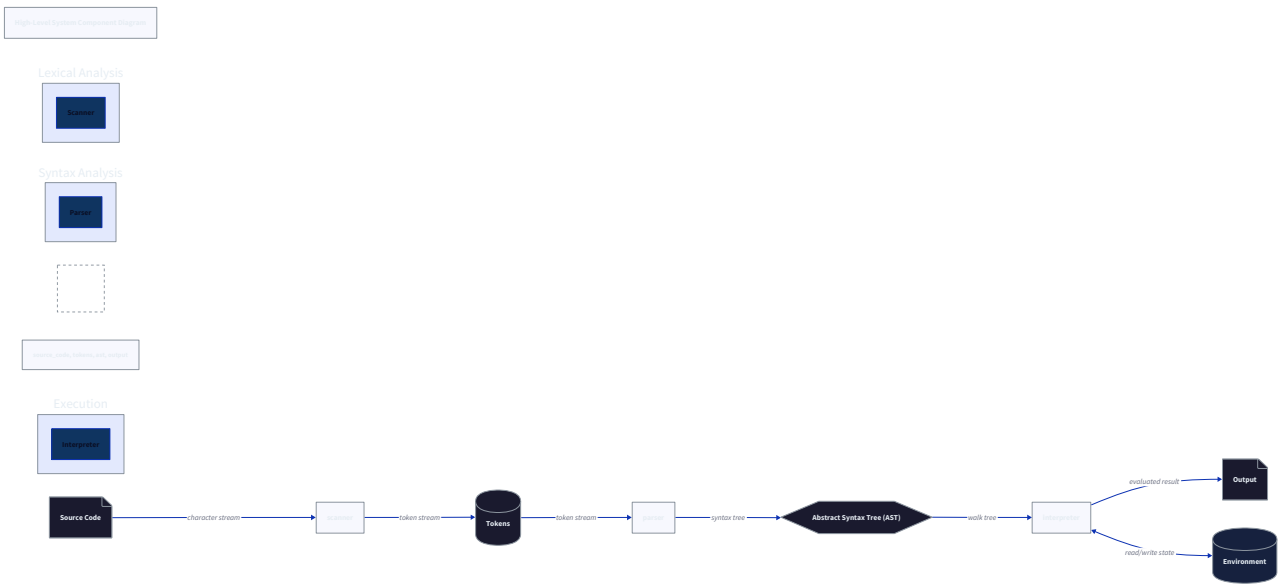
3. High-Level Architecture

This section outlines the macro-architecture of the Lox interpreter, providing a mental map of how source code transforms into executable behavior. The system follows a classic **compiler pipeline** pattern, though it culminates in interpretation rather than code generation. The primary architectural challenge is managing the distinct responsibilities of **static structure analysis** (lexing and parsing) and **dynamic execution** (evaluation), while ensuring state is properly scoped and managed across the lifetime of a program.

Component Overview and Data Flow

At its heart, the interpreter is a linear, **unidirectional pipeline** composed of three core transformational components. Imagine a manufacturing assembly line: raw material (source code) enters at one end, is shaped and assembled through successive stations, and a finished product (program output) emerges at the other. Each station is stateless with respect to the others, consuming the output of the previous station and producing input for the next.

The following diagram illustrates this pipeline and the key data structures that flow between components:



The Pipeline Stages:

1. **Scanner (Lexer) → Tokens:** The Scanner acts as the pipeline's initial quality inspector and disassembler. It consumes a raw string of Lox source code and breaks it down into its smallest meaningful parts, called **Tokens**. This process, known as **lexical analysis**, strips away whitespace and comments, recognizes keywords and operators, and extracts literals (strings and numbers). Its output is a flat, sequential list of `Token` objects, each tagged with a type, the original text (lexeme), the literal value (if any), and its source location.

   > **Design Insight:** The Scanner's job is purely syntactic recognition—it doesn't understand that `var` starts a declaration or that `(` groups an expression. It only knows that the characters `v`, `a`, `r` in sequence form the `VAR` token.

2. **Parser → Abstract Syntax Tree (AST):** The Parser is the assembly line's robotic arm that organizes parts into a structured form. It consumes the linear list of `Token`s and, following Lox's **grammar rules**, builds a hierarchical **Abstract Syntax Tree (AST)**. This tree captures the grammatical nesting and operator precedence inherent in the source code. For example, the expression `1 + 2 * 3` is parsed into a tree where the multiplication node is a child of the addition node, correctly representing that `*` has higher precedence. The parser's output is a `List<Stmt>`, representing the sequence of top-level statements in the program.

   > **Design Insight:** The AST is a *static* representation of the program's *syntax*. It is completely agnostic to runtime values, state, or execution order. It answers "what is the structure of this code?" not "what does this code do when run?"

3. **Interpreter → Program Output/Side Effects:** The Interpreter is the final station where the assembled product is activated. It performs a **tree-walk** over the AST, recursively evaluating each node according to the **semantic rules** of Lox. This is where static syntax meets dynamic behavior: expressions compute values, statements execute commands, and control flow directives direct the walk's path. The interpreter's primary output is side effects: printed text to the console, changes to variable state, or, in a more advanced system, network calls or file I/O.

The Persistent Runtime State: The Environment

While the three core components form a pipeline, the **Interpreter** requires a persistent, dynamic data structure to track program state: the **Environment**. An `Environment` is a scoped mapping from variable names to their current runtime values. It is not a stage in the pipeline but rather a **supporting actor** that the Interpreter consults and modifies during execution.

- **Structure:** Environments form a **chain** via parent (`enclosing`) references. The global scope is the root of this chain. Each new block or function call creates a new child environment. Variable resolution proceeds from the current environment outward toward the global scope.
- **Lifetime:** The Environment chain is built and torn down dynamically at runtime, mirroring the call stack and block structure of the program. This is a key distinction from the AST, which is built once statically.

End-to-End Data Transformation Flow:

The following table traces the complete data transformation for a simple Lox program, `print "Hello, " + "world!";`.

Pipeline Stage	Input	Core Action	Output	Key Data Structure
Scanner	`String`: `'print "Hello, " + "world!";'`	Scan characters, categorize lexemes, ignore whitespace.	`List<Token>`: `[PRINT, STRING("Hello, "), PLUS, STRING("world!"), SEMICOLON, EOF]`	`Token` (type, lexeme, literal, line)
Parser	`List<Token>` (from Scanner)	Recursively group tokens according to grammar rules (e.g., `STRING + STRING` → a `Binary` expression).	`List<Stmt>`: `[PrintStmt(BinaryExpr(Literal("Hello, "), PLUS, Literal("world!")))]`	`Expr` and `Stmt` node hierarchy
Interpreter	`List<Stmt>` (from Parser)	Walk AST: Evaluate `BinaryExpr` (string concatenation) → `"Hello, world!"`, pass to `PrintStmt`.	**Side Effect**: `"Hello, world!"` printed to stdout.	Runtime values (`LoxString`, `LoxNumber`, etc.), `Environment` chain

Key Architectural Decisions:

> **Decision: Tree-Walking Interpreter vs. Bytecode Virtual Machine**
>
> - **Context:** We must choose a strategy for executing Lox programs. The core choice is between directly traversing the AST (tree-walking) or compiling to an intermediate bytecode and executing that on a virtual stack machine.
> - **Options Considered:**
>   1. **Tree-Walking Interpreter:** The interpreter recursively traverses the AST, executing operations at each node.
>   2. **Bytecode Virtual Machine (VM):** A separate compiler phase transforms the AST into a linear sequence of bytecode instructions. A separate VM with a stack and instruction pointer executes this bytecode.
> - **Decision:** Implement a **Tree-Walking Interpreter**.
> - **Rationale:** This is an educational project where clarity and direct mapping from language semantics to implementation are paramount. A tree-walker's control flow mirrors the recursive structure of the language grammar, making it easier to understand and debug. The performance penalty of tree-walking is acceptable for our learning goals.
> - **Consequences:** The interpreter logic is interwoven with the AST traversal (via the Visitor pattern). This simplifies the initial architecture but makes certain optimizations (like direct jumps for loops) more awkward. It also means runtime errors must be propagated via exception handling up the recursive call stack.

Option	Pros	Cons	Chosen?
Tree-Walking Interpreter	Simple, direct mapping from syntax to execution. Easier to implement and debug. No separate compiler/VM phase.	Slower (many pointer indirections). Harder to optimize control flow. Traversal logic can become complex.	**Yes**
Bytecode VM	Faster execution (dense bytecode, tight interpreter loop). Clear separation of compile/run phases. Better foundation for optimizations (JIT).	More complex architecture. Additional concepts (bytecode, stack, IP). Harder to debug execution flow.	No

Decision: Visitor Pattern for AST Operations

- **Context:** We need a clean way to define operations (like evaluation, pretty-printing, static analysis) over our heterogeneous AST node types without resorting to verbose type-checking (`instanceof`) and casting.
- **Options Considered:**
  1. **Visitor Pattern:** Define an interface with a `visit` method for each AST node type. Each node type has an `accept(Visitor)` method that calls the appropriate `visit` method.
  2. **Procedural Switch/ `instanceof` :** Write functions that take a base `Expr` or `Stmt` and use a switch or if-else chain to handle each concrete type.
- **Decision:** Use the **Visitor Pattern**.
- **Rationale:** The Visitor pattern cleanly separates the *structure* of the AST (the node classes) from the *operations* performed on it. This allows us to add new operations (e.g., a static type checker, a code formatter) without modifying the AST node classes themselves (adhering to the Open/Closed Principle). It also makes the type-dispatching logic explicit and compiler-checked.
- **Consequences:** Requires defining two visitor interfaces (`ExprVisitor<V>` , `StmtVisitor<V>`) and adding an `accept` method to every AST node class. This introduces some boilerplate but pays off in maintainability and clarity for the core interpreter operations.

Recommended File/Module Structure

A well-organized codebase is critical for managing the complexity of an interpreter. The following package structure mirrors the pipeline architecture, grouping related components and isolating concerns. This structure is recommended for the primary implementation language, Java.

```
lox/                          # Project root
├── Lox.java                  # Main entry point (orchestrates pipeline)
│
├── scan/                     # Milestone 1: Lexical Analysis
│   ├── Scanner.java          # Converts source string to tokens
│   ├── Token.java            # Token data class
│   └── TokenType.java        # Enumeration of all token types
│
├── parse/                    # Milestones 2 & 3: Syntactic Analysis
│   ├── Parser.java           # Recursive descent parser
│   ├── ParseError.java       # Checked exception for syntax errors
│   │
│   └── ast/                  # Abstract Syntax Tree node definitions
│       ├── Expr.java         # Abstract base class for expressions
│       ├── Stmt.java         # Abstract base class for statements
│       ├── expr/             # Concrete expression nodes
│       │   ├── Binary.java   # left, operator, right
│       │   ├── Unary.java    # operator, right
│       │   ├── Literal.java  # value
│       │   └── ...           # (Grouping, Variable, Assign, etc.)
│       └── stmt/             # Concrete statement nodes
│           ├── Print.java    # expression
│           ├── Var.java      # name, initializer
│           └── ...           # (ExpressionStmt, Block, If, While, etc.)
│
└── interpret/                # Milestones 4-10: Execution
    ├── Interpreter.java      # Tree-walking evaluator (implements ExprVisitor<Object>, StmtVisitor<Void>)
    ├── RuntimeError.java     # Unchecked exception for runtime errors
    │
    ├── environment/          # Runtime state management
    │   └── Environment.java  # Scoped variable storage (values map, enclosing reference)
    │
    └── runtime/              # Representations of Lox values at runtime
        ├── LoxCallable.java  # Interface for callable objects (functions, classes)
        ├── LoxFunction.java  # Represents a Lox function/closure
        ├── LoxClass.java     # Represents a Lox class
        ├── LoxInstance.java  # Represents an instance of a class
        ├── LoxNumber.java    # (Optional) Wrapper for Double
        ├── LoxString.java    # (Optional) Wrapper for String
        ├── LoxBoolean.java   # (Optional) Wrapper for Boolean
        └── LoxNil.java       # Singleton representing nil
```

Package Responsibilities:

- **`lox` (Root):** Contains the driver class `Lox` which coordinates the entire process: reading source (from file or prompt), invoking the scanner, parser, and interpreter, and catching/reporting errors.
- **`scan`:** Isolated lexer module. `Token` and `TokenType` are simple data/enum types used by both the parser and for error reporting.
- **`parse`:** Contains all parsing logic and the immutable AST definition. The `ast` subpackage is a **closed hierarchy**; once defined, new node types should not be added lightly as they require updates to the parser and all visitors.
- **`interpret`:** The heart of runtime execution. It depends on the `parse.ast` package for the tree structure and contains:
  - `Interpreter`: The main engine.
  - `environment`: Manages dynamic scoping.
  - `runtime`: Defines the object model for Lox values. These classes encapsulate Lox semantics (e.g., how `+` works on `LoxString` vs. `LoxNumber`).

Data Flow Between Packages:

The dependencies flow strictly left-to-right, enforcing the pipeline architecture and preventing circular dependencies.

This structure provides a clear roadmap for implementation, allowing you to focus on one logical component at a time, from scanning through to advanced runtime features like classes and inheritance.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option (Consider Later)
Project Build	Plain Java files, compile with `javac`	Use a build system (Maven, Gradle) for dependency and test management
AST Generation	Hand-written visitor pattern classes	Use an annotation processor (e.g., Java Poet) or parser generator (ANTLR) to generate boilerplate
Runtime Value Representation	Use plain Java types (`Double` , `String` , `Boolean`) with `null` for nil	Wrap in dedicated classes (e.g., `LoxNumber`) for explicit type tagging and custom behavior
Error Reporting	Print formatted messages to `System.err`	Use a structured logging library (SLF4J) or collect errors for IDE integration

B. Recommended File/Module Structure (Starter)

Create the following directory and empty Java files to establish the project skeleton. This enforces the architectural separation from the start.

```
mkdir -p lox/scan lox/parse/ast/expr lox/parse/ast/stmt lox/interpret/environment lox/interpret/runtime
```

C. Infrastructure Starter Code

The following are complete, foundational classes that you can use as-is. They define the core data structures that flow between components.

File: `lox/scan/TokenType.java`

```java
package scan;

// Exhaustive enum of all token types in Lox.
// Based on Chapter 4 of Crafting Interpreters.
public enum TokenType {
    // Single-character tokens.
    LEFT_PAREN, RIGHT_PAREN, LEFT_BRACE, RIGHT_BRACE,
    COMMA, DOT, MINUS, PLUS, SEMICOLON, SLASH, STAR,

    // One or two character tokens.
    BANG, BANG_EQUAL,
    EQUAL, EQUAL_EQUAL,
    GREATER, GREATER_EQUAL,
    LESS, LESS_EQUAL,

    // Literals.
    IDENTIFIER, STRING, NUMBER,

    // Keywords.
    AND, CLASS, ELSE, FALSE, FUN, FOR, IF, NIL, OR,
    PRINT, RETURN, SUPER, THIS, TRUE, VAR, WHILE,

    EOF
}
```

File: `lox/scan/Token.java`

```java
package scan;

// Represents a single lexical token from the source code.
public class Token {
    public final TokenType type;
    public final String lexeme;    // The raw text of the token
    public final Object literal;   // The interpreted value for literals (String, Double, null)
    public final int line;         // Source line number (1-indexed) for error reporting

    public Token(TokenType type, String lexeme, Object literal, int line) {
        this.type = type;
        this.lexeme = lexeme;
        this.literal = literal;
        this.line = line;
    }

    public String toString() {
        return type + " " + lexeme + " " + literal;
    }
}
```

File: `lox/interpret/environment/Environment.java`

JAVA

```java
package interpret.environment;

import java.util.HashMap;
import java.util.Map;

// A chain of scopes mapping variable names to values.
// This is a core runtime data structure for state management.
public class Environment {
    // The immediately enclosing scope. `null` for the global environment.
    public final Environment enclosing;
    // Storage for variables defined in this specific scope.
    private final Map<String, Object> values = new HashMap<>();

    // Constructor for the global scope (no enclosing environment).
    public Environment() {
        this.enclosing = null;
    }

    // Constructor for a new nested scope.
    public Environment(Environment enclosing) {
        this.enclosing = enclosing;
    }

    // Defines a new variable in the current scope.
    // Used for 'var' declarations.
    public void define(String name, Object value) {
        values.put(name, value);
    }

    // Gets the value of a variable, searching outward through enclosing scopes.
    // Throws a runtime error if the variable is not found.
    public Object get(Token name) {
        if (values.containsKey(name.lexeme)) {
            return values.get(name.lexeme);
        }
        // Look up in the parent scope.
        if (enclosing != null) return enclosing.get(name);

        throw new RuntimeError(name, "Undefined variable '" + name.lexeme + "'.");
```

```
    }

    // Assigns a new value to an existing variable.
    // Searches outward through enclosing scopes. Throws if variable not found.
    public void assign(Token name, Object value) {
        if (values.containsKey(name.lexeme)) {
            values.put(name.lexeme, value);
            return;
        }
        // Try to assign in the parent scope.
        if (enclosing != null) {
            enclosing.assign(name, value);
            return;
        }
        throw new RuntimeError(name, "Undefined variable '" + name.lexeme + "'.");
    }
}
```

D. Core Logic Skeleton Code

File: `lox/Lox.java` **(Main Driver)**

JAVA

```java
package lox;

import scan.Scanner;

import scan.Token;

import parse.Parser;

import parse.ast.Stmt;

import interpret.Interpreter;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.List;

// The main entry point. Can run files or a REPL.

public class Lox {

    // Static interpreter instance shared across runs in REPL mode.

    private static final Interpreter interpreter = new Interpreter();

    // Flag to indicate if a parsing or runtime error occurred.

    static boolean hadError = false;

    static boolean hadRuntimeError = false;

    public static void main(String[] args) throws IOException {

        if (args.length > 1) {

            System.out.println("Usage: jlox [script]");

            System.exit(64); // EX_USAGE

        } else if (args.length == 1) {

            runFile(args[0]);

        } else {

            runPrompt();

        }

    }

    // Read and execute a Lox source file.

    private static void runFile(String path) throws IOException {

        byte[] bytes = Files.readAllBytes(Paths.get(path));

        run(new String(bytes, "UTF-8"));
```

```
        // Indicate an error in the exit code.
        if (hadError) System.exit(65); // EX_DATAERR
        if (hadRuntimeError) System.exit(70); // EX_SOFTWARE
    }

    // Start an interactive Read-Eval-Print Loop (REPL).
    private static void runPrompt() throws IOException {
        InputStreamReader input = new InputStreamReader(System.in);
        BufferedReader reader = new BufferedReader(input);

        for (;;) {
            System.out.print("> ");
            String line = reader.readLine();
            if (line == null) break; // Ctrl-D
            run(line);
            // Reset error flags for the next line in the REPL.
            hadError = false;
            hadRuntimeError = false;
        }
    }

    // Core run routine: scan, parse, interpret.
    private static void run(String source) {
        // TODO 1: Instantiate the Scanner with the source string.
        // TODO 2: Call scanner.scanTokens() to get the token list.
        // TODO 3: Instantiate the Parser with the token list.
        // TODO 4: Call parser.parse() to get the statement list.
        // TODO 5: If there were no syntax errors (hadError is false), call interpreter.interpret(statements).
        // Hint: Catch ParseError and RuntimeError exceptions and report them using the error() and runtimeError()
methods below.
    }

    // Error reporting helpers (to be called from Scanner, Parser, Interpreter).
    static void error(int line, String message) {
        report(line, "", message);
    }
    static void error(Token token, String message) {
        if (token.type == TokenType.EOF) {
```

```
            report(token.line, " at end", message);
        } else {
            report(token.line, " at '" + token.lexeme + "'", message);
        }
    }
    private static void report(int line, String where, String message) {
        System.err.println("[line " + line + "] Error" + where + ": " + message);
        hadError = true;
    }
    static void runtimeError(RuntimeError error) {
        System.err.println(error.getMessage() + "\n[line " + error.token.line + "]");
        hadRuntimeError = true;
    }
}
```

E. Language-Specific Hints (Java)

- **Visitor Pattern Implementation:** Use generic interfaces `ExprVisitor<R>` and `StmtVisitor<R>` with a generic return type `R`. This allows the interpreter to return `Object` (values) and `Void` (for statements).
- **Runtime Value Representation:** Java's `Double`, `String`, and `Boolean` can be used directly, with `null` representing Lox's `nil`. For type checking, use `instanceof` (e.g., `if (leftOperand instanceof Double)`).
- **Immutability:** Make AST node fields `final` to ensure they are immutable after construction. This prevents accidental modification and simplifies reasoning.
- **Error Handling:** Use a custom `ParseError` (extends `RuntimeException`) for syntax errors to enable panic-mode recovery in the parser. Use a custom `RuntimeError` (also extends `RuntimeException`) for runtime errors like type mismatches.

F. Milestone Checkpoint (Architecture Verification)

After setting up the skeleton structure and the `Lox` driver class, you can verify the pipeline is plumbed correctly with a simple test:

1. **Create a test file:** `test.lox` containing a single line: `print "Hello, world!";`
2. **Run the driver:** `java lox.Lox test.lox`
3. **Expected Output at this stage:** The program should compile without errors. When run, it may do nothing (if the scanner/parser/interpreter are stubs) or throw a `NullPointerException`. The key is that the project structure is in place and the main class runs without compilation errors.
4. **Next Step:** The first real output will come after implementing the **Scanner (Milestone 1)**, when you can print the list of tokens to verify lexing works.

4. Data Model

> **Milestone(s):** All milestones (foundational data structures)

This section defines the core immutable data structures that form the backbone of our interpreter. Think of these as the **permanent artifacts** produced and consumed as source code moves through the interpretation pipeline. Unlike the procedural logic of scanning, parsing, or evaluating, these data structures are the **static definitions** that represent the program at each phase of its lifecycle. Properly designing these structures is crucial because they determine how information flows between components and what capabilities our interpreter can support.

The data model has four interrelated layers:

1. **Tokens** – The atomic units of meaning from the source text
2. **AST Nodes** – The hierarchical tree structure representing program syntax
3. **Runtime Values** – The living data that exists during program execution
4. **Environments** – The scoped namespace that maps variable names to runtime values

Each layer builds upon the previous one, creating a clear separation between static structure and dynamic execution. This separation is fundamental to the interpreter's architecture—it allows us to reason about the program's syntax independently from its runtime behavior.

Tokens: The Lexical Atoms

Mental Model: The Scrabble Tiles

> Think of tokens as Scrabble tiles formed from the continuous stream of source code letters. The scanner's job is to break the text into these discrete, categorized tiles—some represent numbers (`5.2`), others operators (`+`), keywords (`var`), or punctuation (`;`). Each tile carries not just its face value (the actual text) but also its classification (is this tile an addition operator or a string concatenator?). These tiles are then passed to the parser, which assembles them into meaningful structures according to grammatical rules.

Tokens are the smallest meaningful units produced by the **Scanner** (Milestone 1). They represent the categorized lexical elements of the Lox language, stripping away insignificant whitespace and comments while preserving the essential structure of the source code.

Token Structure Every token contains four pieces of information that collectively identify what it represents and where it came from:

Field Name	Type	Description
`type`	`TokenType`	The category of token (e.g., `PLUS` , `IDENTIFIER` , `NUMBER`). Determines how the parser will interpret this token.
`lexeme`	`String`	The actual text from the source code that generated this token. For the source `123.45` , the lexeme would be `"123.45"` .
`literal`	`Object`	The runtime value associated with the token, if any. For a number token, this would be a `Double` ; for a string token, a `String` ; for identifiers and keywords, `null` .
`line`	`int`	The line number in the source file where this token begins (1-indexed). Critical for error reporting.

TokenType Enumeration The `TokenType` enumeration defines all possible token categories in Lox. Each token type corresponds to a specific lexical pattern in the language grammar. The complete set includes:

Category	Examples	Notes
Single-character tokens	`LEFT_PAREN` , `RIGHT_PAREN` , `LEFT_BRACE` , `RIGHT_BRACE` , `COMMA` , `DOT` , `MINUS` , `PLUS` , `SEMICOLON` , `SLASH` , `STAR`	These map directly to individual characters in source code.
One-or-two character tokens	`BANG` , `BANG_EQUAL` , `EQUAL` , `EQUAL_EQUAL` , `GREATER` , `GREATER_EQUAL` , `LESS` , `LESS_EQUAL`	The scanner must look ahead to distinguish `!` from `!=` .
Literals	`IDENTIFIER` , `STRING` , `NUMBER`	These tokens carry additional value in their `literal` field.
Keywords	`AND` , `CLASS` , `ELSE` , `FALSE` , `FUN` , `FOR` , `IF` , `NIL` , `OR` , `PRINT` , `RETURN` , `SUPER` , `THIS` , `TRUE` , `VAR` , `WHILE`	Reserved words that cannot be used as identifiers.
End-of-file	`EOF`	A special sentinel token indicating the end of the input stream.

> **Design Insight:** The `literal` field uses Java's `Object` type (which can be `null`) rather than a dedicated union type because it simplifies the implementation while maintaining type safety through careful programming. For number literals, we store `Double`; for string literals, `String`; for `true` / `false` / `nil` keywords, we use the corresponding runtime value objects (which we'll define later). This design allows the scanner to produce values that can flow directly into the interpreter without conversion.

Token Lifecycle

1. **Creation**: The scanner creates tokens as it recognizes lexical patterns in the source text. For example, when it encounters the sequence `"hello"`, it creates a `Token` with `type=STRING`, `lexeme="\"hello\""`, `literal="hello"` (without quotes), and `line` set to the current line.
2. **Consumption**: The parser receives the stream of tokens and uses their `type` fields to guide parsing decisions. The `literal` field may be used to embed constant values directly into the AST.
3. **Error Reporting**: When a parse or runtime error occurs, the associated token's `line` (and potentially `lexeme`) is used to generate a user-friendly error message like `"Error at line 5: Unexpected token '}'"`.

ADR: Token Representation Strategy

> **Decision: Unified Token Class with Optional Literal Field**
>
> - **Context**: We need to represent categorized lexical elements with associated metadata (source location, actual text, and sometimes a computed value).
> - **Options Considered**:
>   1. **Separate token classes for each type**: Create distinct `IdentifierToken`, `NumberToken`, `OperatorToken` etc., each with type-specific fields.
>   2. **Tagged union/discriminated record**: Use a single `Token` type with an enum tag and type-specific data in a union (not natively supported in Java).
>   3. **Unified class with optional fields**: One `Token` class with fields for all possible metadata, some of which may be `null`.
> - **Decision**: Use a single `Token` class with all four fields, where `literal` may be `null` for tokens without associated values.
> - **Rationale**: This approach is simple to implement in Java, minimizes class explosion, and matches the typical Java pattern for token representation. The occasional `null` checks are acceptable given the small number of token types that carry literals. It also keeps the scanner and parser logic straightforward.
> - **Consequences**: We get a clean, immutable token representation. However, we must be careful to handle `null` literals appropriately in the parser and interpreter. Type safety is maintained through discipline rather than the type system.

Option	Pros	Cons	Chosen?
Separate token classes	Type-safe, no null checks	Class explosion, harder to handle generically	No
Tagged union	Type-safe, memory efficient	Not natively supported in Java, requires pattern matching	No
Unified class	Simple, fewer classes, easy to pass around	Some fields null for certain tokens, less type-safe	**Yes**

AST Nodes: The Program's Skeleton

Mental Model: The Russian Nesting Dolls

> The Abstract Syntax Tree (AST) is like a set of Russian nesting dolls, where each doll contains smaller dolls inside it. A `Binary` expression doll contains two smaller expression dolls (left and right operands) plus an operator token. A `Block` statement doll contains a list of statement dolls inside it. The parser's job is to assemble these nested structures by matching grammatical patterns in the token stream, creating a complete hierarchical representation that mirrors the program's syntactic structure.

AST nodes form the **canonical representation** of a Lox program's syntax after parsing (Milestones 2-3). Unlike concrete syntax trees (which include every detail like parentheses and semicolons), the AST abstracts away syntactic noise, leaving only the essential structure needed for execution.

Node Hierarchy The AST is composed of two parallel class hierarchies: `Expr` for expressions (which produce values) and `Stmt` for statements (which perform actions). Both are abstract base classes that define the Visitor pattern interface.

Expression Node Types Each expression node type represents a distinct syntactic construct that evaluates to a value:

Node Type	Key Fields	Description	Example Lox Code
`Binary`	`Expr left`, `Token operator`, `Expr right`	Binary operation with two operands and an operator token.	`1 + 2`
`Unary`	`Token operator`, `Expr right`	Unary operation with one operand and an operator token.	`-5` or `!true`
`Grouping`	`Expr expression`	Parenthesized expression (for explicit precedence).	`(1 + 2) * 3`
`Literal`	`Object value`	Constant literal value (number, string, boolean, nil).	`42`, `"hello"`, `true`, `nil`
`Variable`	`Token name`	Reference to a variable by its name token.	`x`
`Assign`	`Token name`, `Expr value`	Assignment to a previously declared variable.	`x = 10`
`Logical`	`Expr left`, `Token operator`, `Expr right`	Logical `and` or `or` with short-circuit evaluation.	`a and b`
`Call`	`Expr callee`, `Token paren`, `List<Expr> arguments`	Function or method call with argument list.	`foo(1, 2)`
`Get`	`Expr object`, `Token name`	Property access on an instance using dot notation.	`obj.property`
`Set`	`Expr object`, `Token name`, `Expr value`	Property assignment on an instance.	`obj.property = 5`
`This`	`Token keyword`	Reference to the current instance within a method.	`this`
`Super`	`Token keyword`, `Token method`	Reference to a superclass method.	`super.method()`

Statement Node Types Statement nodes represent syntactic constructs that perform actions but don't produce values (except for expression statements, which evaluate an expression for side effects):

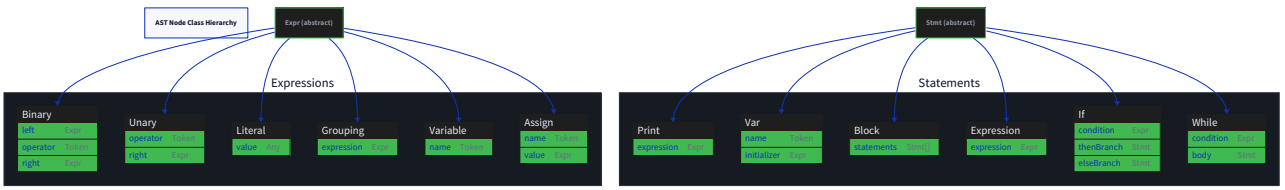
Node Type	Key Fields	Description	Example Lox Code
`Expression`	`Expr expression`	Expression evaluated for side effects.	`x + 1;`
`Print`	`Expr expression`	Evaluates expression and prints result.	`print "hello";`
`Var`	`Token name`, `Expr initializer`	Variable declaration with optional initializer.	`var x = 5;`
`Block`	`List<Stmt> statements`	Block of statements creating a new scope.	`{ var x = 1; print x; }`
`If`	`Expr condition`, `Stmt thenBranch`, `Stmt elseBranch`	Conditional execution (else branch may be null).	`if (x) print "yes"; else print "no";`
`While`	`Expr condition`, `Stmt body`	Loop with precondition.	`while (x < 10) x = x + 1;`
`Function`	`Token name`, `List<Token> params`, `List<Stmt> body`	Function declaration (name, parameters, body).	`fun add(a, b) { return a + b; }`
`Return`	`Token keyword`, `Expr value`	Return statement from function (value may be null).	`return 42;`
`Class`	`Token name`, `Expr.Variable superclass`, `List<Stmt.Function> methods`	Class declaration with optional superclass.	`class Point { init(x, y) { this.x = x; this.y = y; } }`

Visitor Pattern Architecture The Visitor pattern enables **open recursion**—allowing new operations on the AST without modifying the node classes themselves. This is essential because our interpreter will define multiple operations: pretty-printing, static analysis (in advanced implementations), and interpretation.

> **Design Insight:** The Visitor pattern creates a double dispatch mechanism. When we call `accept(visitor)` on an AST node, the node calls the appropriate `visitXxx` method on the visitor, passing itself as an argument. This gives the visitor type-safe access to the node's specific fields without requiring instanceof checks.

AST Immutability All AST nodes are **immutable**—their fields are set at construction and never modified. This has several benefits:

1. **Thread safety**: Though our interpreter is single-threaded, immutability prevents accidental modification.
2. **Predictable behavior**: An AST can be traversed multiple times without fear of side effects.
3. **Simpler reasoning**: Once parsed, the program structure doesn't change during execution.



ADR: Visitor Pattern vs. Instanceof Checks

> **Decision: Visitor Pattern for AST Operations**
>
> - **Context**: We need to define operations (like evaluation, pretty-printing) that work across the heterogeneous AST node types.
> - **Options Considered**:
>   1. **Instanceof checks with casting**: Use a switch on node type with explicit casts in each operation.
>   2. **Interpreter method in each node**: Put an `interpret()` method directly in each AST node class.
>   3. **Visitor pattern**: Separate the operations from the node structure using double dispatch.
> - **Decision**: Use the Visitor pattern with separate `Expr.Visitor<R>` and `Stmt.Visitor<V>` interfaces.
> - **Rationale**: The Visitor pattern cleanly separates concerns—node definitions are purely structural, while operations are defined elsewhere. This makes it easy to add new operations (like a static analyzer or code generator) without modifying the AST classes. It's also the pattern used in the reference "Crafting Interpreters" book, providing consistency for learners.
> - **Consequences**: We incur some boilerplate (the `accept()` method in each node), but gain flexibility and organization. The pattern also naturally supports return values from visits (unlike the interpreter-in-each-node approach).

Option	Pros	Cons	Chosen?
Instanceof checks	Simple, no boilerplate	Type-unsafe, spreads logic across type checks	No
Interpreter in each node	Encapsulated, no casting	Mixes structure and behavior, hard to add operations	No
Visitor pattern	Separates concerns, extensible, type-safe	Boilerplate accept methods, more complex	**Yes**

Runtime Values: The Interpreter's Currency

Mental Model: The Theater Props

> Runtime values are like props in a theater production. When the interpreter "performs" the program (walks the AST), it needs tangible objects to work with: numbers to calculate, strings to concatenate, function objects to call, and instances to manipulate. Each value type has specific capabilities (a function can be called, a class can be instantiated) just as each prop has a specific purpose (a sword can be swung, a book can be opened). These props are created, passed around, and transformed during the performance.

Runtime values are the **living data** that exist during program execution (Milestones 4-10). They flow through the interpreter as expressions evaluate, get stored in variables, and are passed between functions. The interpreter's entire purpose is to produce and manipulate these values according to the semantics defined by the AST structure.

Value Type Hierarchy Lox is dynamically typed, meaning values carry their type at runtime. We represent this in Java using a hierarchy of classes, all ultimately extending `Object`. However, for clarity and to avoid confusion with Java's `Object`, we'll refer to these as **Lox values**.

Value Type	Java Representation	Description	Example Literals
LoxNumber	`Double`	64-bit floating-point number (IEEE 754).	`123`, `4.56`, `-7.89`
LoxString	`String`	Immutable sequence of Unicode characters.	`"hello"`, `"multi\nline"`
LoxBoolean	`Boolean`	Logical truth value.	`true`, `false`
LoxNil	`null` sentinel	Represents absence of a value.	`nil`
LoxFunction	`LoxFunction` class	Callable function object with parameters, body, and closure environment.	Created by `fun` declarations
LoxClass	`LoxClass` class	Callable class object that creates instances and holds methods.	Created by `class` declarations
LoxInstance	`LoxInstance` class	Instance of a class with its own field storage.	Created by class constructor calls

Truthiness Rules Lox defines **truthiness** (whether a value is considered "true" in boolean contexts) as:

- `false` and `nil` are **falsy**
- **Everything else** is **truthy** (including `0`, empty strings, and even `0.0`)

This differs from some languages but matches Lox's semantics: only explicit falsehoods are falsy.

Value Operations Each value type supports specific operations:

Operation	Supported Types	Behavior
Arithmetic (`+`, `-`, `*`, `/`)	`LoxNumber`	Standard math. `+` also works for string concatenation.
Comparison (`<`, `<=`, `>`, `>=`)	`LoxNumber`	Numeric ordering.
Equality (`==`, `!=`)	All types	Value equality (structural for objects, reference for functions/classes/instances).
Negation (`-`)	`LoxNumber`	Unary minus.
Logical NOT (`!`)	Any	Returns `true` if operand is falsy, `false` otherwise.
Call (`(...)`)	`LoxFunction`, `LoxClass`	Invokes with arguments.
Property access (`.`)	`LoxInstance`	Gets/sets field values or calls methods.

ADR: Unified vs. Tagged Value Representation

> **Decision: Polymorphic Value Classes with Common Supertype**
>
> - **Context**: We need to represent Lox values in Java, which is statically typed, while Lox is dynamically typed.
> - **Options Considered**:
>   1. **Single `Object` with instanceof checks**: Store all values as Java `Object` and check types at runtime.
>   2. **Tagged union/enum wrapper**: Create a `Value` class with an enum tag and type-specific storage.
>   3. **Polymorphic hierarchy**: Create specific classes for each Lox type, all implementing a common `LoxValue` interface.
> - **Decision**: Use specific classes (`LoxFunction`, `LoxClass`, `LoxInstance`) for complex types, and Java built-in types (`Double`, `String`, `Boolean`) for simple ones, with `null` for nil.
> - **Rationale**: This approach minimizes boilerplate while leveraging Java's type system where possible. Built-in types give us arithmetic and comparison operators for free. For user-defined types (functions, classes, instances), we need custom behavior (calling, instantiating, property access), so classes are natural. Using `null` for nil is idiomatic in Java.
> - **Consequences**: We must be careful with `null` checks (nil is a valid Lox value, not an error). Some operations require explicit type checking (e.g., `+` for numbers vs. strings). The interpreter's `evaluate()` method returns `Object`, which we must cast appropriately.

Option	Pros	Cons	Chosen?
Single `Object`	Simple, uniform	Lots of instanceof, no type safety	No
Tagged union	Explicit type tags, can add new types easily	Manual tag checking, memory overhead	No
Polymorphic hierarchy	Natural for OOP, leverages Java types	Mixed representation (built-ins + custom classes)	**Yes**

Environments: The Scoped Namespace

Mental Model: The Russian Doll Scopes

> Environments are like a set of nested Russian dolls, where each doll represents a scope. The innermost doll (current scope) can see its own contents and everything in the dolls enclosing it, but outer dolls cannot see inside inner ones. When you enter a function or block, you open a new doll inside the current one; when you exit, you close it and return to the outer doll. Variables are stored in the smallest doll that contains their declaration.

Environments implement **lexical scoping** (Milestones 5, 8) by maintaining a chain of variable bindings. Each environment corresponds to a scope in the program: global scope, function scope, block scope, or class method scope. The environment chain forms a tree structure that the interpreter traverses when resolving variable names.

Environment Structure Each environment is essentially a dictionary (name → value) with a reference to its parent (enclosing) environment:

Field Name	Type	Description
`enclosing`	`Environment`	The parent environment (null for the global scope).
`values`	`Map<String, Object>`	The variable bindings in this scope (name → runtime value).

Core Operations Environments support three fundamental operations:

Method	Parameters	Returns	Description
`define`	`String name`, `Object value`	`void`	Creates a new variable in the current environment. Used for `var` declarations.
`get`	`Token name`	`Object`	Looks up a variable by its name token, searching outward through enclosing environments. Throws if not found.
`assign`	`Token name`, `Object value`	`void`	Updates an existing variable's value, searching outward. Throws if variable wasn't previously defined.

Environment Chains in Practice Consider this Lox program:

```lox
var global = "outside";
{
  var inner = "inside";
  print global; // Looks in inner scope, then outer (global) scope
}
```

This creates the following environment structure:

```
Global Environment {global: "outside"}
    ↑
Block Environment {inner: "inside"}
```

When the interpreter evaluates `print global;` inside the block:

1. It looks for `"global"` in the block environment → not found
2. It follows the `enclosing` link to the global environment
3. It finds `"global"` there and returns its value `"outside"`

Closure Environment Capture For closures (Milestone 8), the key insight is that a function **captures its defining environment** (the environment active when the function was created, not when it's called). This captured environment becomes the parent of the function's call environment:

```
Global {x: 10}
    ↑ (captured)
makeCounter's defining environment
    ↑
call environment (when makeCounter() is called)
```

This allows the inner function to access `x` even after `makeCounter` has returned.

ADR: Environment Chain vs. Static Distance

Decision: Simple Parent-Chain Lookup

- **Context**: We need to resolve variable names to their values at runtime, respecting lexical scoping rules.
- **Options Considered**:
  1. **Parent-chain lookup**: Each environment stores a reference to its parent; lookups walk the chain.
  2. **Static distance/indices**: At compile time, compute the "distance" (number of hops) to the defining environment; at runtime, navigate directly.
  3. **Flat closure representation**: Copy all captured variables into the closure at creation time.
- **Decision**: Use parent-chain lookup for simplicity and clarity.
- **Rationale**: Parent-chain lookup is straightforward to implement and understand—it directly models the mental model of nested scopes. While less efficient than static distance (which requires a separate compilation phase), it's sufficient for an educational interpreter. It also makes closures simple: a function just stores a reference to its defining environment.
- **Consequences**: Variable lookup is O(depth) where depth is the nesting level. For deeply nested code, this could be slow, but Lox programs in practice are shallow. The design also naturally supports dynamic additions to outer scopes (though Lox doesn't allow this).

Option	Pros	Cons	Chosen?
Parent-chain lookup	Simple, intuitive, easy closures	O(depth) lookup time	**Yes**
Static distance	O(1) lookup, efficient	Requires compile-time analysis, more complex	No
Flat closures	Fast access to captured variables	Complex copy semantics, doesn't support mutation of outer variables	No

Common Pitfalls with Environments

⚠️ Pitfall: Forgetting to Create New Environment for Each Function Call

- **Description**: Reusing the same environment for multiple calls to the same function.
- **Why it's wrong**: Function parameters and local variables from previous calls would persist, breaking recursion and causing incorrect behavior.
- **Fix**: Every function call must create a **fresh** environment with the function's closure environment as its parent.

⚠️ Pitfall: Incorrect Parent Link in Block Environments

- **Description**: Setting a block environment's parent to the global environment instead of the current environment.
- **Why it's wrong**: The block wouldn't have access to variables in enclosing functions or outer blocks.
- **Fix**: When creating a block environment, pass the **current** environment as the parent.

⚠️ Pitfall: Not Checking for Undefined Variables in assign()

- **Description**: Allowing assignment to create new variables implicitly (like Python) instead of requiring explicit declaration.
- **Why it's wrong**: Lox requires `var` declarations; assignment to undefined variables should be a runtime error.
- **Fix**: In `Environment.assign()`, search through the chain and throw if the name isn't found.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Data Structures	Java Collections (`HashMap`, `ArrayList`)	Custom persistent/immutable collections
Value Representation	Java built-in types + custom classes	Tagged union with explicit type hierarchy
Environment Lookup	Linear chain search	Compile-time static distance resolution

Recommended File/Module Structure

```
lox/
├── Token.java              # Token class definition
├── TokenType.java          # TokenType enum
├── ast/
│   ├── Expr.java           # Expression base class and visitor
│   ├── Stmt.java           # Statement base class and visitor
│   └── (concrete AST node classes can be inner classes or separate files)
├── runtime/
│   ├── LoxFunction.java    # Function value
│   ├── LoxClass.java       # Class value
│   ├── LoxInstance.java    # Instance value
│   └── Environment.java    # Environment chain
└── Interpreter.java        # Main interpreter (evaluates AST)
```

Infrastructure Starter Code

Complete, ready-to-use code for foundational data structures:

JAVA

```java
// Token.java

package lox;


import java.util.Objects;


public class Token {

    public final TokenType type;

    public final String lexeme;

    public final Object literal;

    public final int line;


    public Token(TokenType type, String lexeme, Object literal, int line) {

        this.type = type;

        this.lexeme = lexeme;

        this.literal = literal;

        this.line = line;

    }


    @Override

    public String toString() {

        return type + " " + lexeme + " " + literal;

    }


    @Override

    public boolean equals(Object o) {

        if (this == o) return true;

        if (o == null || getClass() != o.getClass()) return false;

        Token token = (Token) o;

        return line == token.line &&

               type == token.type &&

               Objects.equals(lexeme, token.lexeme) &&

               Objects.equals(literal, token.literal);

    }


    @Override

    public int hashCode() {

        return Objects.hash(type, lexeme, literal, line);

    }
```

```
}
```

JAVA

```java
// TokenType.java

package lox;

public enum TokenType {

    // Single-character tokens

    LEFT_PAREN, RIGHT_PAREN, LEFT_BRACE, RIGHT_BRACE,

    COMMA, DOT, MINUS, PLUS, SEMICOLON, SLASH, STAR,

    // One or two character tokens

    BANG, BANG_EQUAL,

    EQUAL, EQUAL_EQUAL,

    GREATER, GREATER_EQUAL,

    LESS, LESS_EQUAL,

    // Literals

    IDENTIFIER, STRING, NUMBER,

    // Keywords

    AND, CLASS, ELSE, FALSE, FUN, FOR, IF, NIL, OR,

    PRINT, RETURN, SUPER, THIS, TRUE, VAR, WHILE,

    EOF

}
```

JAVA

```java
// Environment.java (starter version - will be extended)

package lox.runtime;


import java.util.HashMap;

import java.util.Map;


import lox.Token;

import lox.Lox;


public class Environment {

    final Environment enclosing;

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


    public Environment() {

        this.enclosing = null;

    }


    public Environment(Environment enclosing) {

        this.enclosing = enclosing;

    }


    public void define(String name, Object value) {

        values.put(name, value);

    }


    public Object get(Token name) {

        if (values.containsKey(name.lexeme)) {

            return values.get(name.lexeme);

        }


        if (enclosing != null) return enclosing.get(name);


        throw new RuntimeError(name,

            "Undefined variable '" + name.lexeme + "'.");

    }


    public void assign(Token name, Object value) {

        if (values.containsKey(name.lexeme)) {

            values.put(name.lexeme, value);

            return;
```

```
        }

        if (enclosing != null) {
            enclosing.assign(name, value);
            return;
        }

        throw new RuntimeError(name,
            "Undefined variable '" + name.lexeme + "'.");
    }
}
```

Core Logic Skeleton Code

AST node base classes with Visitor pattern:

JAVA

```java
// ast/Expr.java

package lox.ast;


import java.util.List;

import lox.Token;


public abstract class Expr {

    public interface Visitor<R> {

        R visitBinaryExpr(Binary expr);

        R visitUnaryExpr(Unary expr);

        R visitGroupingExpr(Grouping expr);

        R visitLiteralExpr(Literal expr);

        R visitVariableExpr(Variable expr);

        R visitAssignExpr(Assign expr);

        R visitLogicalExpr(Logical expr);

        R visitCallExpr(Call expr);

        R visitGetExpr(Get expr);

        R visitSetExpr(Set expr);

        R visitThisExpr(This expr);

        R visitSuperExpr(Super expr);

    }


    public abstract <R> R accept(Visitor<R> visitor);


    public static class Binary extends Expr {

        public final Expr left;

        public final Token operator;

        public final Expr right;


        public Binary(Expr left, Token operator, Expr right) {

            this.left = left;

            this.operator = operator;

            this.right = right;

        }


        @Override

        public <R> R accept(Visitor<R> visitor) {

            return visitor.visitBinaryExpr(this);
```

```
        }
    }

    // TODO: Add other expression node classes (Unary, Grouping, Literal,
    // Variable, Assign, Logical, Call, Get, Set, This, Super) following
    // the same pattern as Binary above.
    // Each should have:
    // 1. Public final fields for its data
    // 2. A constructor initializing those fields
    // 3. An accept() method calling the appropriate visitor method
}
```

JAVA

```java
// ast/Stmt.java

package lox.ast;


import java.util.List;

import lox.Token;


public abstract class Stmt {

    public interface Visitor<V> {

        V visitExpressionStmt(Expression stmt);

        V visitPrintStmt(Print stmt);

        V visitVarStmt(Var stmt);

        V visitBlockStmt(Block stmt);

        V visitIfStmt(If stmt);

        V visitWhileStmt(While stmt);

        V visitFunctionStmt(Function stmt);

        V visitReturnStmt(Return stmt);

        V visitClassStmt(Class stmt);

    }


    public abstract <V> V accept(Visitor<V> visitor);


    public static class Expression extends Stmt {

        public final Expr expression;


        public Expression(Expr expression) {

            this.expression = expression;

        }


        @Override

        public <V> V accept(Visitor<V> visitor) {

            return visitor.visitExpressionStmt(this);

        }

    }


    // TODO: Add other statement node classes (Print, Var, Block, If,

    // While, Function, Return, Class) following the same pattern.

    // Remember: FunctionStmt should store name (Token), params (List<Token>),

    // and body (List<Stmt>).
```

```
}
```

Language-Specific Hints

1. **Use `final` fields** in all data classes (Token, AST nodes) to ensure immutability.
2. **Override `toString()`** in Token and AST nodes for debugging.
3. **Use `HashMap` for environments** - it's simple and fast enough for our needs.
4. **Be careful with `null`** - Lox's `nil` is represented by Java `null`, but not all `null` values are Lox nil (could be uninitialized Java references).
5. **Use `Double` for numbers** - Lox uses double-precision floats. Remember that `1 + 2` produces `3.0` (a Double), not `3` (an Integer).

Milestone Checkpoint: Data Structures

After implementing the data model, you should be able to:

- Compile all Java files without errors
- Create Token objects manually in a test and verify their fields
- Construct a simple AST manually (e.g., `new Expr.Binary(new Expr.Literal(1), new Token(TokenType.PLUS, "+", null, 1), new Expr.Literal(2))`)
- Create nested environments and verify variable lookup follows the chain

Run a simple test:

JAVA

```java
// Test.java

public class Test {

    public static void main(String[] args) {

        // Test Token

        Token plus = new Token(TokenType.PLUS, "+", null, 1);

        System.out.println("Token: " + plus);



        // Test Environment chain

        Environment global = new Environment();

        global.define("x", 10.0);



        Environment local = new Environment(global);

        local.define("y", 20.0);



        Token xToken = new Token(TokenType.IDENTIFIER, "x", null, 1);

        Token yToken = new Token(TokenType.IDENTIFIER, "y", null, 1);



        System.out.println("x in local: " + local.get(xToken)); // Should print 10.0

        System.out.println("y in local: " + local.get(yToken)); // Should print 20.0

    }

}
```

Expected output:

```
Token: PLUS + null
x in local: 10.0
y in local: 20.0
```

If you get `NullPointerException` or incorrect values, check:

1. Environment chain links (parent references)
2. HashMap key matching (names are case-sensitive)
3. That you're using `Double` values, not `Integer`

> **Milestone(s):** Milestone 1 - Scanner (Lexer)

5.1 Component Design: Scanner (Lexer) [Milestone 1]

The **Scanner**, also called the **lexer** (lexical analyzer), is the gateway through which raw Lox source code enters the interpreter pipeline. Its singular responsibility is to transform a linear sequence of characters into a structured sequence of **tokens**—meaningful atomic units that represent the basic vocabulary of the language. Without this component, the interpreter would see only an undifferentiated stream of characters, unable to distinguish keywords from identifiers or numbers from operators.

Mental Model: The Tokenizer

Think of the scanner as a **language translator breaking down a sentence into labeled dictionary entries**. Given the sentence "The quick brown fox jumps over 42.5 lazy dogs," a translator would identify each word ("The" → article, "quick" → adjective, "brown" → adjective, "fox" → noun, "jumps" → verb, etc.), recognize "42.5" as a number, and note the period as punctuation. The scanner performs exactly this role for Lox source code: it reads character by character, recognizes patterns that constitute valid language elements, categorizes them (is "while" a keyword or an identifier?), and outputs a structured list where each entry knows its type, exact text, literal value (for numbers and strings), and position in the source. This tokenized form provides the parser with manageable, semantically meaningful units instead of raw characters.

Interface and State

The Scanner is a stateful component that processes source code in a single, linear pass. Its interface is deliberately minimal: a constructor accepting the source string and a single public method that drives the scanning process.

Scanner Public Interface

Method	Parameters	Returns	Description
`Scanner` constructor	`source: String`	`Scanner` instance	Creates a new scanner for the given source code string. Stores the source and initializes scanning state.
`scanTokens`	None	`List<Token>`	The main entry point. Scans the entire source, returning a complete list of tokens. This method orchestrates the scanning loop until the end of the source is reached.

Scanner Internal State

The scanner maintains four essential pieces of state during its operation, each tracked by an integer index or counter:

Field Name	Type	Description
`source`	`String`	The complete source code text to be scanned. This is the input character stream.
`tokens`	`List<Token>`	The accumulating list of successfully scanned tokens. This is the output of the scanning process.
`start`	`int`	Index in `source` pointing to the first character of the *current* token being scanned.
`current`	`int`	Index in `source` pointing to the *next* character to be examined (the "lookahead" position).
`line`	`int`	Current line number in the source (1-indexed). Incremented when newline characters are encountered. Essential for accurate error reporting.

The relationship between `start` and `current` is fundamental: `start` marks where the current token began, while `current` marches forward through the source as characters are examined. When a complete token is recognized, the substring from `start` to `current-1` becomes the token's `lexeme`, and both positions are captured in a new `Token` object added to the `tokens` list. Then `start` is reset to `current` to begin the next token.

Scanning Algorithm

The scanner operates via a central loop that repeatedly examines characters, identifies token boundaries, and emits tokens until the source is exhausted. The algorithm follows a deterministic, procedural flow with clear branching based on the character at the current position.

Step-by-Step Scanning Procedure

1. **Initialization**: Create empty token list. Set `start = 0`, `current = 0`, `line = 1`.
2. **Main Scanning Loop**: While not at end of source (`current < source.length()`): a. **Reset token start**: Set `start = current`. b. **Examine next character**: Call `peek()` to look at character at `current` without consuming it. c. **Branch on character type**:
   - **Single-character tokens** (`(` , `)` , `{` , `}` , `,` , `.` , `-` , `+` , `;` , `*` , `/`): Directly create corresponding token.
   - **One-or-two-character tokens** (`!` , `=` , `<` , `>` , `==` , `!=` , `<=` , `>=`): Check next character for `=` to decide between single or double operator.
   - **String literals** (`"`): Enter string scanning mode, consuming until closing `"` (handling escapes).
   - **Number literals** (`0` - `9`): Enter number scanning mode, consuming digits and optional decimal point followed by more digits.
   - **Identifiers and keywords** (`a` - `z` , `A` - `Z` , `_`): Enter identifier scanning mode, consuming alphanumeric/underscore characters, then check if lexeme matches a reserved keyword.
   - **Whitespace** (` ` , `\t` , `\r` , `\n`): Skip (update line count for newlines).
   - **Comments** (`//` or `/*`): Skip until end of line or closing `*/` .
   - **Unrecognized character**: Report lexical error with line number. d. **Advance position**: Increment `current` as characters are consumed. e. **Emit token**: For recognized tokens (excluding whitespace/comments), create `Token` with type, lexeme, literal value (if any), and current line, then add to `tokens` .
3. **Finalization**: After loop, add an `EOF` (end-of-file) token to mark the end of the token stream.

Detailed Scanning Logic for Key Token Types

String Literals:

1. Advance past opening `"` .
2. While next character is not `"` :
   - If end of file, report error: "Unterminated string."
   - If newline, increment `line` (Lox supports multiline strings).
   - If `\` (backslash), process escape sequence: `\"` , `\\` , `\n` , `\t` , etc.
   - Otherwise, add character to string value.
3. Advance past closing `"` .
4. Create token with literal value as the accumulated string (without quotes).

Number Literals:

1. Consume consecutive digits (`0` - `9`).
2. If next character is `.` and character after that is a digit:
   - Consume `.`.
   - Consume consecutive digits after decimal.
3. Convert lexeme to `double` value (Lox uses double-precision floating point).

Identifiers and Keywords:

1. Consume consecutive alphanumeric characters and underscores (`a` - `z` , `A` - `Z` , `0` - `9` , `_`).
2. Check if resulting lexeme matches a reserved keyword (e.g., "and", "class", "if", "while").
   - Use a hash map from string to `TokenType` for efficient lookup.
3. If match, emit keyword token; otherwise, emit identifier token.

ADR: Visitor vs. Procedural Scanning

> **Decision: Procedural Scanning Loop Over State Machine/Visitor Patterns**
>
> - **Context**: The scanner must reliably recognize a finite set of token patterns from a linear character stream. While lexical analysis can be implemented using various formalisms (state machines, table-driven scanners, or visitor patterns), we need a solution that is educational, straightforward to implement and debug, and maps clearly to the lexical grammar of Lox.
> - **Options Considered**:
>   1. **Procedural scanning loop**: A manual loop with explicit character-by-character examination and branching via `switch` / `if` statements.
>   2. **Finite-state machine (FSM)**: Explicit state transitions driven by character classes, possibly implemented via state pattern or transition table.
>   3. **Visitor pattern scanning**: Separate visitor classes for each major token category that traverse the character stream.
> - **Decision**: Implement a **procedural scanning loop**.
> - **Rationale**:
>   1. **Educational clarity**: The procedural approach maps directly to the human mental model of "look at character, decide what to do." Each token type's recognition logic is localized and explicit, making it easier for learners to trace execution and understand the scanning process.
>   2. **Simplicity**: No need to define state classes, transition tables, or visitor hierarchies. The scanner's logic is contained in one straightforward class with helper methods.
>   3. **Direct control**: Error reporting and recovery (like skipping invalid characters) can be inserted precisely where needed in the flow.
>   4. **Performance**: While not a primary goal, the direct character manipulation and explicit branching are efficient for the small-scale scanning needs of an educational interpreter.
> - **Consequences**:
>   - **Positive**: Implementation is straightforward to write, read, and debug. Changes to token recognition (e.g., adding a new operator) involve localized edits.
>   - **Negative**: The scanner class becomes a "god object" containing all scanning logic, which could grow large. However, for Lox's limited token set, this is manageable.
>   - **Mitigation**: Logic for different token categories can be separated into well-named private methods (e.g., `scanString()` , `scanNumber()` , `scanIdentifier()`).

Options Comparison Table

Option	Pros	Cons	Why Not Chosen
Procedural scanning loop	Direct mapping to lexical grammar; Easy to understand and debug; Simple error handling; No abstraction overhead	Can become monolithic; Manual character-by-character handling may be verbose	**Chosen** - Best aligns with educational goals and simplicity
Finite-state machine	Formal, clean separation of states; Potentially easier to extend for complex patterns	Significant boilerplate for state classes; Obfuscates the straightforward logic; Harder to debug state transitions	Adds unnecessary complexity for Lox's simple lexical patterns
Visitor pattern scanning	Separates token recognition logic into visitor classes; Follows OOP principles	Heavy abstraction for a linear process; Visitors don't naturally fit scanning; Overkill for simple token patterns	The visitor pattern is better suited for heterogeneous tree traversal (AST), not linear scanning

Common Pitfalls

⚠️ Pitfall: Mishandling Multi-Character Operators

Description: Failing to properly handle operators that could be one or two characters long (`=`, `==`, `!`, `!=`, `<`, `<=`, `>`, `>=`). A naive implementation might emit a single `=` token when seeing `==`, leaving the second `=` to be scanned as a separate token or causing a parse error.

Why it's wrong: The parser expects a single `==` token for equality comparison. Receiving two separate `=` tokens would lead to a syntax error or incorrect parsing (e.g., interpreting `a == b` as assignment `a = = b`).

How to avoid: Implement a `match(expected)` helper that looks at the next character. If it matches `expected`, consume it and return `true`. Use this when scanning `=`, `!`, `<`, `>`: if `match('=')` returns `true`, emit the two-character operator; otherwise emit the single-character one.

⚠️ Pitfall: Unterminated Strings Without Error Recovery

Description: When scanning a string literal, if the closing double quote is missing (e.g., `"hello`), the scanner may either crash (index out of bounds) or enter an infinite loop consuming the rest of the source.

Why it's wrong: Lexical errors should be reported gracefully with location information, not cause interpreter crashes. The scanner should detect the EOF before the closing quote and report a meaningful error.

How to avoid: In `scanString()`, check for EOF (`peek() == '\0'`) at each iteration. If EOF is reached before the closing quote, call an error reporting function with the line number and message "Unterminated string." Then break out of the loop to continue scanning the rest of the source (or halt, depending on error recovery strategy).

⚠️ Pitfall: Incorrect Number Literal Scanning

Description: Numbers with multiple decimal points (e.g., `123.45.67`) might be incorrectly tokenized as a single number, or numbers trailing with a dot (e.g., `123.`) might be mishandled.

Why it's wrong: These are invalid numeric literals in Lox. The scanner should emit a single token for valid numbers but should not silently accept invalid ones. A number with two decimal points is a syntax error that should be caught by the scanner.

How to avoid: After consuming the integer part and seeing a `.`, peek at the *next* character. If it's a digit, consume the dot and fractional digits. If it's not a digit, **do not consume the dot**—the number ends at the previous digit, and the dot will be scanned as a separate `.` token (likely leading to a parse error). This correctly handles both `123.` (two tokens: `123` and `.`) and `123.45` (one token).

⚠️ Pitfall: Forgetting Line Number Tracking

Description: Not updating the `line` counter when encountering newline characters (`\n`), especially inside strings or when skipping comments.

Why it's wrong: Error messages (lexical, syntactic, runtime) rely on accurate line numbers to point developers to the problematic code. If line numbers are off by even one, debugging becomes significantly harder.

How to avoid: Increment `line` consistently in three places: 1) When advancing past a `\n` character in the main scanning loop (for newlines in general code). 2) Inside `scanString()` when a `\n` is encountered (Lox supports multiline strings). 3) Inside block comment skipping when `\n` is encountered.

⚠️ Pitfall: Keyword Matching Case-Sensitivity Issues

Description: Using case-sensitive string comparison for keywords when Lox keywords are case-sensitive (`while` is a keyword, `While` is not).

Why it's wrong: If matching is case-insensitive, `While` would be incorrectly tokenized as a keyword rather than an identifier, preventing users from using that as a variable name (contrary to language spec).

How to avoid: Use exact string equality when checking the identifier lexeme against the keyword map. Lox's keywords are all lowercase, so the lexeme must match exactly in case.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Scanner Implementation	Manual character-by-character scanning with `String.charAt()`	Using `java.io.Reader` with buffering for file I/O, or regex for token patterns (not recommended for learning)
Token Storage	`ArrayList<Token>`	`LinkedList<Token>` if frequent insertion at middle needed (not the case here)
Keyword Lookup	`HashMap<String, TokenType>` with static initialization	Trie data structure for efficient prefix matching (overkill for ~20 keywords)

B. Recommended File/Module Structure

```
lox/
├── Lox.java                 # Main entry point, coordinates components
├── Token.java              # Token class definition
├── TokenType.java          # Enumeration of token types
├── scanner/
│   ├── Scanner.java        # Scanner implementation (this component)
│   └── ScannerTest.java    # Unit tests for scanner
├── parser/                 # (Milestone 2-3)
├── ast/                    # (Milestone 2)
└── interpreter/            # (Milestone 4+)
```

C. Infrastructure Starter Code

The `Token` class and `TokenType` enum are foundational data structures used throughout the interpreter. Here is their complete implementation:

TokenType.java (complete):

```java
package com.craftinginterpreters.lox;

// Each enum value represents a distinct category of lexical element in Lox.
public enum TokenType {
    // Single-character tokens.
    LEFT_PAREN, RIGHT_PAREN, LEFT_BRACE, RIGHT_BRACE,
    COMMA, DOT, MINUS, PLUS, SEMICOLON, SLASH, STAR,

    // One or two character tokens.
    BANG, BANG_EQUAL,
    EQUAL, EQUAL_EQUAL,
    GREATER, GREATER_EQUAL,
    LESS, LESS_EQUAL,

    // Literals.
    IDENTIFIER, STRING, NUMBER,

    // Keywords.
    AND, CLASS, ELSE, FALSE, FUN, FOR, IF, NIL, OR,
    PRINT, RETURN, SUPER, THIS, TRUE, VAR, WHILE,

    // End-of-file marker.
    EOF
}
```

Token.java (complete):

```java
package com.craftinginterpreters.lox;


// Represents a single lexical token with its metadata.

public class Token {

    public final TokenType type;    // Category of token (e.g., NUMBER, WHILE)

    public final String lexeme;     // Raw text as it appeared in source

    public final Object literal;    // Runtime value for literals (String, Double)

    public final int line;          // Line number where token starts (1-indexed)


    public Token(TokenType type, String lexeme, Object literal, int line) {

        this.type = type;

        this.lexeme = lexeme;

        this.literal = literal;

        this.line = line;

    }


    @Override

    public String toString() {

        return type + " " + lexeme + " " + literal;

    }

}
```

D. Core Logic Skeleton Code

Scanner.java (skeleton with TODOs):

JAVA

```java
package com.craftinginterpreters.lox.scanner;

import com.craftinginterpreters.lox.Token;

import com.craftinginterpreters.lox.TokenType;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class Scanner {

    private final String source;          // The raw source code

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

    private int start = 0;                // Start index of current token

    private int current = 0;              // Current character index being examined

    private int line = 1;                 // Current line number

    // Static map for keyword lookup: lexeme -> TokenType

    private static final Map<String, TokenType> keywords;

    static {

        keywords = new HashMap<>();

        keywords.put("and",    TokenType.AND);

        keywords.put("class",  TokenType.CLASS);

        keywords.put("else",   TokenType.ELSE);

        keywords.put("false",  TokenType.FALSE);

        keywords.put("for",    TokenType.FOR);

        keywords.put("fun",    TokenType.FUN);

        keywords.put("if",     TokenType.IF);

        keywords.put("nil",    TokenType.NIL);

        keywords.put("or",     TokenType.OR);

        keywords.put("print",  TokenType.PRINT);

        keywords.put("return", TokenType.RETURN);

        keywords.put("super",  TokenType.SUPER);

        keywords.put("this",   TokenType.THIS);

        keywords.put("true",   TokenType.TRUE);

        keywords.put("var",    TokenType.VAR);

        keywords.put("while",  TokenType.WHILE);

    }
```

```java
public Scanner(String source) {
    this.source = source;
}

// Main entry point: scan all tokens from source.
public List<Token> scanTokens() {
    // TODO 1: Loop while not at end of source
    while (!isAtEnd()) {
        // We are at the beginning of the next token.
        start = current;
        scanToken();
    }

    // TODO 2: Add EOF token at the end
    tokens.add(new Token(TokenType.EOF, "", null, line));
    return tokens;
}

// Examines the next character(s) and emits a token.
private void scanToken() {
    // TODO 3: Get the next character (consume it)
    char c = advance();


    // TODO 4: Branch based on character using switch statement
    switch (c) {
        // Single-character tokens
        case '(': addToken(TokenType.LEFT_PAREN); break;
        case ')': addToken(TokenType.RIGHT_PAREN); break;
        case '{': addToken(TokenType.LEFT_BRACE); break;
        case '}': addToken(TokenType.RIGHT_BRACE); break;
        case ',': addToken(TokenType.COMMA); break;
        case '.': addToken(TokenType.DOT); break;
        case '-': addToken(TokenType.MINUS); break;
        case '+': addToken(TokenType.PLUS); break;
        case ';': addToken(TokenType.SEMICOLON); break;
        case '*': addToken(TokenType.STAR); break;
```

```
// One-or-two-character tokens
case '!':
    // TODO 5: If next char is '=', emit BANG_EQUAL else BANG
    addToken(match('=') ? TokenType.BANG_EQUAL : TokenType.BANG);
    break;
case '=':
    // TODO 6: If next char is '=', emit EQUAL_EQUAL else EQUAL
    addToken(match('=') ? TokenType.EQUAL_EQUAL : TokenType.EQUAL);
    break;
case '<':
    // TODO 7: If next char is '=', emit LESS_EQUAL else LESS
    addToken(match('=') ? TokenType.LESS_EQUAL : TokenType.LESS);
    break;
case '>':
    // TODO 8: If next char is '=', emit GREATER_EQUAL else GREATER
    addToken(match('=') ? TokenType.GREATER_EQUAL : TokenType.GREATER);
    break;

// Division operator or comment
case '/':
    // TODO 9: If next char is '/', it's a single-line comment
    if (match('/')) {
        // Consume until end of line
        while (peek() != '\n' && !isAtEnd()) advance();
    }
    // TODO 10: Optional - handle block comments /* ... */
    // else if (match('*')) {
    //     blockComment();
    // }
    else {
        addToken(TokenType.SLASH);
    }
    break;
```

```
            // Whitespace (ignore, but track newlines)
            case ' ':
            case '\r':
            case '\t':
                break;
            case '\n':
                // TODO 11: Increment line counter
                line++;
                break;

            // String literals
            case '"':
                // TODO 12: Call string() method to handle string scanning
                string();
                break;

            default:
                // TODO 13: Handle numbers (if c is digit)
                if (isDigit(c)) {
                    number();
                }
                // TODO 14: Handle identifiers/keywords (if c is letter or underscore)
                else if (isAlpha(c)) {
                    identifier();
                }
                else {
                    // TODO 15: Report lexical error for unexpected character
                    Lox.error(line, "Unexpected character: '" + c + "'");
                }
                break;
        }
    }

    // Scans a string literal: assumes opening " has been consumed.
    private void string() {
        // TODO 16: Loop while next char is not '"' and not at end
```

```
        while (peek() != '"' && !isAtEnd()) {
            // TODO 17: If newline, increment line counter (Lox supports multiline strings)
            if (peek() == '\n') line++;
            // TODO 18: Handle escape sequences if desired (optional milestone)
            advance();
        }

        // TODO 19: Check for unterminated string (if at end)
        if (isAtEnd()) {
            Lox.error(line, "Unterminated string.");
            return;
        }

        // TODO 20: Consume the closing "
        advance();

        // TODO 21: Extract string value (without quotes) and add token
        String value = source.substring(start + 1, current - 1);
        addToken(TokenType.STRING, value);
    }

    // Scans a number literal: digits optionally followed by . and more digits.
    private void number() {
        // TODO 22: Consume consecutive digits
        while (isDigit(peek())) advance();

        // TODO 23: Look for fractional part (decimal point followed by digits)
        if (peek() == '.' && isDigit(peekNext())) {
            // Consume the decimal point
            advance();
            // Consume fractional digits
            while (isDigit(peek())) advance();
        }

        // TODO 24: Convert lexeme to double and add NUMBER token
        double value = Double.parseDouble(source.substring(start, current));
```

```
        addToken(TokenType.NUMBER, value);
    }

    // Scans an identifier or keyword: alphanumeric characters and underscores.
    private void identifier() {
        // TODO 25: Consume alphanumeric characters and underscores
        while (isAlphaNumeric(peek())) advance();

        // TODO 26: Check if lexeme is a keyword
        String text = source.substring(start, current);
        TokenType type = keywords.get(text);
        if (type == null) type = TokenType.IDENTIFIER;

        // TODO 27: Add appropriate token (keyword or identifier)
        addToken(type);
    }

    // =============== Helper Methods ===============

    // Returns true if we've consumed all characters.
    private boolean isAtEnd() {
        return current >= source.length();
    }

    // Consumes and returns the next character.
    private char advance() {
        current++;
        return source.charAt(current - 1);
    }

    // Returns the next character without consuming it (lookahead).
    private char peek() {
        if (isAtEnd()) return '\0';
        return source.charAt(current);
    }

    // Returns the character after the next (two-character lookahead).
    private char peekNext() {
```

```
        if (current + 1 >= source.length()) return '\0';
        return source.charAt(current + 1);
    }

    // Checks if the next character matches expected, consuming it only if true.
    private boolean match(char expected) {
        // TODO 28: If at end, return false
        if (isAtEnd()) return false;
        // TODO 29: If next character matches expected, consume it and return true
        if (source.charAt(current) != expected) return false;
        current++;
        return true;
    }

    // Adds a token with no literal value.
    private void addToken(TokenType type) {
        addToken(type, null);
    }

    // Adds a token with a literal value.
    private void addToken(TokenType type, Object literal) {
        // TODO 30: Extract lexeme from source and create token
        String text = source.substring(start, current);
        tokens.add(new Token(type, text, literal, line));
    }

    // Returns true if c is a digit (0-9).
    private boolean isDigit(char c) {
        return c >= '0' && c <= '9';
    }

    // Returns true if c is a letter (a-z, A-Z) or underscore.
    private boolean isAlpha(char c) {
        return (c >= 'a' && c <= 'z') ||
               (c >= 'A' && c <= 'Z') ||
                c == '_';
    }
```

```
    // Returns true if c is alphanumeric or underscore.
    private boolean isAlphaNumeric(char c) {
        return isAlpha(c) || isDigit(c);
    }
}
```

Note on Error Reporting: The skeleton references `Lox.error(line, message)`. You'll need to implement a simple error reporting mechanism in your main `Lox` class. For now, you can use:

```java
public class Lox {
    static void error(int line, String message) {
        report(line, "", message);
    }

    private static void report(int line, String where, String message) {
        System.err.println("[line " + line + "] Error" + where + ": " + message);
    }
}
```

E. Language-Specific Hints

- **Character Access**: Use `String.charAt()` for simple character access. For production code, consider converting the source to a `char[]` for performance, but for clarity, `charAt()` is fine.
- **String Substrings**: `source.substring(start, current)` creates a new string each time. This is acceptable for scanning, but be aware of memory if scanning huge files.
- **Number Parsing**: `Double.parseDouble()` handles both integer and decimal numbers. Lox uses double-precision floating point for all numbers.
- **HashMap Initialization**: The static initializer for `keywords` runs once when the class is loaded. This is efficient and thread-safe for this use case.
- **Error Handling**: For lexical errors, you might want to collect multiple errors rather than halting at the first. Consider adding an `errors` list to the scanner and reporting all at once.

F. Milestone Checkpoint

After implementing the scanner, you should be able to test it with simple Lox code snippets:

Test Command (assuming you have a simple main method in `Lox.java` that creates a scanner and prints tokens):

```bash
cd /path/to/lox

javac com/craftinginterpreters/lox/*.java com/craftinginterpreters/lox/scanner/*.java

java com.craftinginterpreters.lox.Lox "var x = 42;"
```

Expected Output: A list of tokens printed, something like:

```
VAR var null
IDENTIFIER x null
EQUAL = null
NUMBER 42 42.0
SEMICOLON ; null
EOF  null
```

Verification Steps:

1. **Single-character tokens**: Test `( ) { } , . - + ; * /` all produce correct tokens.
2. **Multi-character operators**: Test `= == ! != < <= > >=` produce correct single or double tokens.
3. **Strings**: Test `"hello"` produces STRING token with literal "hello". Test `"multi\nline"` handles escape sequences (optional) and newline counting.
4. **Numbers**: Test `123`, `123.456`, `.456` (should be DOT then NUMBER 456), `123.` (should be NUMBER 123 then DOT).
5. **Keywords and identifiers**: Test `while` → WHILE keyword, `While` → IDENTIFIER.
6. **Whitespace and comments**: Test that spaces, tabs, and `// comments` are ignored and don't produce tokens.
7. **Error reporting**: Test `"unterminated` produces "Unterminated string" error with line number. Test `@` produces "Unexpected character" error.

Signs of Problems:

- Infinite loop: Likely missing advance() calls or incorrect condition in scanning loop.
- Missing tokens: `start` and `current` not being reset properly after token emission.
- Incorrect line numbers: Not incrementing `line` on newlines inside strings or comments.
- Number parsing errors: `Double.parseDouble()` throwing exception for invalid numbers like `123.` (should be caught by your logic before parsing).

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Scanner goes into infinite loop	Missing `advance()` call in a branch, or `isAtEnd()` always returns false	Add debug prints showing `current` and character at each loop iteration.	Ensure every path through `scanToken()` either advances `current` or breaks/returns.
String literal includes quotes in value	Using `start` to `current` without trimming quotes	Print the extracted substring value before creating token.	In `string()` method, use `start + 1` and `current - 1` as indices.
Numbers with decimal point produce two tokens	Not checking that character after `.` is a digit before consuming	Print lexeme when scanning numbers. Check logic for fractional part.	Use `peekNext()` to look ahead two characters when seeing `.`.
Keywords not recognized	Case mismatch or missing entry in keywords map	Print the extracted identifier text and check map lookup.	Ensure keyword map has all lowercase entries and identifier is converted to lowercase for lookup (if language is case-insensitive; Lox is case-sensitive).
Line numbers off by one	Not counting newlines in strings or block comments	Add print statement showing line number for each token.	Increment `line` counter when encountering `\n` in `string()` and comment-skipping logic.
"Unterminated string" error on valid strings	Not consuming the closing quote character	Check that `advance()` is called after the while loop in `string()`.	Ensure the closing `"` is consumed before creating the token.

> **Milestone(s):** Milestone 2 - Representing Code (AST), Milestone 3 - Parsing Expressions

5.2 Component Design: Parser & AST [Milestones 2 & 3]

This section details the design of the **Parser**, the component that translates a linear sequence of tokens into a structured, hierarchical **Abstract Syntax Tree (AST)**, and the definition of the AST node classes themselves. If the scanner breaks a paragraph into words, the parser is the grammarian who diagrams the sentence, identifying the subject, verb, object, and how they relate through nesting and precedence rules. This transformation from `List<Token>` to `List<Stmt>` is the heart of syntactic analysis, bridging the gap between raw text and executable structure.

Mental Model: The Sentence Diagrammer

Imagine you are given a jumbled list of words and punctuation from a sentence: `["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "."]`. Your task is to reconstruct the sentence's grammatical structure. You identify "fox" as the subject noun, "jumps" as the verb, and "over the lazy dog" as a prepositional phrase modifying the verb. You then arrange these into a tree showing that "quick brown" modifies "fox", and "lazy" modifies "dog".

The **Parser** performs exactly this role for programming languages. It consumes the flat list of tokens produced by the scanner and applies the formal grammar rules of Lox to build a tree structure (the AST). This tree explicitly represents the nesting (parentheses, blocks), precedence (multiplication before addition), and associativity (left-to-right for most operators) that are only implicit in the linear token stream. Each node in the tree corresponds to a syntactic construct in the language—an expression, a statement, a declaration—and its children are the sub-components of that construct.

AST Definition and the Visitor Pattern

The **Abstract Syntax Tree (AST)** is the canonical, in-memory representation of a Lox program's syntactic structure after parsing. It is "abstract" because it omits syntactic details that don't affect meaning, like the exact placement of parentheses (though their grouping effect is captured) and semicolons between statements. The AST serves as the immutable input to all subsequent phases: the **Interpreter** walks it to execute the program, and a **PrettyPrinter** can traverse it to regenerate formatted source code.

The AST is defined as a hierarchy of node types, rooted in two abstract base classes: `Expr` for expressions (which produce a value) and `Stmt` for statements (which perform an action). Each concrete node type (e.g., `Binary`, `If`) is a subclass of one of these and holds fields referencing its child nodes and relevant tokens.

AST Node Type (Class)	Base Class	Key Fields (Name → Type)	Description
Binary	`Expr`	`left` → `Expr`, `operator` → `Token`, `right` → `Expr`	A binary operation like `1 + 2` or `x == y`.
Unary	`Expr`	`operator` → `Token`, `right` → `Expr`	A unary operation like `-5` or `!true`.
Grouping	`Expr`	`expression` → `Expr`	A parenthesized expression `(1 + 2)`.
Literal	`Expr`	`value` → `Object`	A literal value: number, string, boolean, `nil`.
Variable	`Expr`	`name` → `Token`	A reference to a variable by its name.
Assign	`Expr`	`name` → `Token`, `value` → `Expr`	An assignment `x = 5`.
Logical	`Expr`	`left` → `Expr`, `operator` → `Token`, `right` → `Expr`	A logical `and` or `or` expression.
Call	`Expr`	`callee` → `Expr`, `paren` → `Token`, `arguments` → `List<Expr>`	A function call `fn(1, 2)`.
Get	`Expr`	`object` → `Expr`, `name` → `Token`	Property access `obj.property`.
Set	`Expr`	`object` → `Expr`, `name` → `Token`, `value` → `Expr`	Property assignment `obj.property = 5`.
This	`Expr`	`keyword` → `Token`	The `this` keyword.
Super	`Expr`	`keyword` → `Token`, `method` → `Token`	A superclass method call `super.method()`.
Expression	`Stmt`	`expression` → `Expr`	A statement that wraps a single expression.
Print	`Stmt`	`expression` → `Expr`	A `print` statement.
Var	`Stmt`	`name` → `Token`, `initializer` → `Expr`	A `var` declaration.
Block	`Stmt`	`statements` → `List<Stmt>`	A block `{ ... }` of statements.
If	`Stmt`	`condition` → `Expr`, `thenBranch` → `Stmt`, `elseBranch` → `Stmt`	An `if` statement with optional `else`.
While	`Stmt`	`condition` → `Expr`, `body` → `Stmt`	A `while` loop.
Function	`Stmt`	`name` → `Token`, `params` → `List<Token>`, `body` → `List<Stmt>`	A `fun` function declaration.
Return	`Stmt`	`keyword` → `Token`, `value` → `Expr`	A `return` statement.
Class	`Stmt`	`name` → `Token`, `superclass` → `Expr.Variable`, `methods` → `List<Stmt.Function>`	A `class` declaration.

To perform operations over this heterogeneous tree (like evaluation or printing), we need a way to dispatch to type-specific logic for each node. A naive approach uses a cascade of `instanceof` checks and casts, which is brittle and verbose. The **Visitor Pattern** provides an elegant, type-safe alternative. It leverages **double dispatch**: the AST node's `accept` method calls the appropriate `visit` method on the visitor object, passing itself as an argument. This allows the visitor to define behavior for each node type in a single class.

> **Key Insight:** The Visitor Pattern effectively adds a virtual method to each AST node type without modifying the node classes themselves. This keeps the AST data structures simple and immutable, while allowing arbitrary operations (interpretation, printing, static analysis) to be defined as separate, cohesive visitor classes.

The pattern requires two visitor interfaces—one for expressions, one for statements—each declaring a `visit` method for every concrete node type.

Interface Method	Parameters	Returns	Description
`Expr.Visitor<R>.visitBinaryExpr`	`Binary` expr	`R`	Visit a `Binary` expression node.
`Expr.Visitor<R>.visitUnaryExpr`	`Unary` expr	`R`	Visit a `Unary` expression node.
... (and so on for every `Expr` subclass)	...	...	...
`Stmt.Visitor<V>.visitExpressionStmt`	`Expression` stmt	`V`	Visit an `Expression` statement node.
`Stmt.Visitor<V>.visitPrintStmt`	`Print` stmt	`V`	Visit a `Print` statement node.
... (and so on for every `Stmt` subclass)	...	...	...

Every concrete `Expr` and `Stmt` node implements an `accept` method that calls the corresponding visitor method, passing `this`.

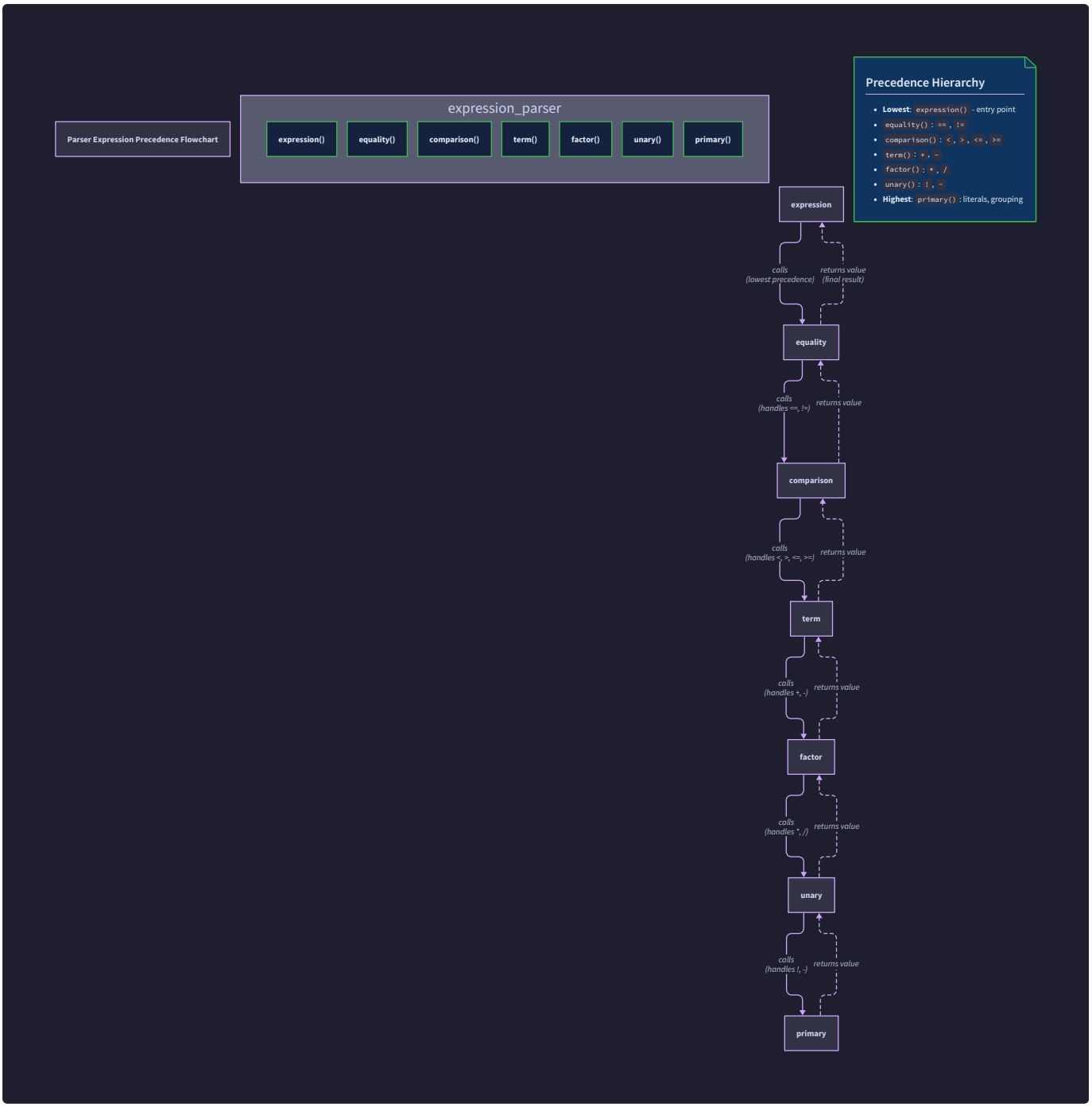
Parsing Expression Grammar (PEG) & Recursive Descent

The parser must implement the **grammar rules** that define valid Lox programs. We use a **Parsing Expression Grammar (PEG)** style, which is well-suited for **recursive descent parsing**. In recursive descent, each grammar rule becomes a function in the parser. The parser's internal state is simple: a list of tokens and an index pointing to the next token to consume (`current`).

The core challenge is correctly parsing expressions with multiple levels of **operator precedence** and **associativity**. Lox has the following precedence levels (from lowest/binding loosest to highest/binding tightest):

1. **Assignment** (`=`), which is right-associative.
2. **Logical `or`** (`or`), left-associative.
3. **Logical `and`** (`and`), left-associative.
4. **Equality** (`==` , `!=`), left-associative.
5. **Comparison** (`<` , `>` , `<=` , `>=`), left-associative.
6. **Term** (`+` , `-`), left-associative.
7. **Factor** (`*` , `/`), left-associative.
8. **Unary** (`!` , `-`), right-associative.
9. **Primary** (literals, variables, grouping, calls, etc.).

The recursive descent parser encodes this precedence hierarchy directly into its call graph. The function for the lowest-precedence operator (`assignment`) calls the function for the next higher level (`logical_or`), which calls the next (`logical_and`), and so on, until reaching `primary()` , which handles atomic expressions. This creates a natural nesting where higher-precedence operators are parsed deeper in the recursive call stack, resulting in an AST where they are lower in the tree (closer to the leaves).



The parsing algorithm for a single expression follows this step-by-step procedure:

1. **Start Parsing**: The entry point `expression()` (or `assignment()` for Lox) is called.
2. **Parse Lower Precedence First**: The `assignment()` method first calls `logicalOr()` to parse the left-hand side. This begins the descent down the precedence chain.
3. **Recursive Descent**: Control flows down through `logicalOr()` → `logicalAnd()` → `equality()` → `comparison()` → `term()` → `factor()` → `unary()` → `primary()`. Each function is responsible for parsing its own level of operators and delegating sub-expressions to the function for the next higher precedence level.
4. **Handle Operators at Current Level**: When a function (e.g., `term()`) regains control after its delegate call (to `factor()`), it checks if the current token is an operator at its level (`+` or `-`). If so, it consumes the operator, recursively calls `factor()` again to parse the right operand, and builds a `Binary` AST node. It repeats in a loop to handle chains of same-precedence operators (e.g., `1 + 2 - 3`), ensuring left associativity.
5. **Build AST**: As the recursion unwinds, each function returns an `Expr` node representing the parsed sub-expression at its precedence level. The `Binary` and `Unary` nodes constructed become the children of nodes at lower precedence levels.

6. **Final Result**: The top-level `assignment()` function may perform additional logic (like checking for `=` for assignment) and finally returns the complete `Expr` AST for the entire expression.

For statements, the parsing is more straightforward, as statements do not have complex precedence interactions. The `parse()` method of the parser repeatedly calls `statement()` until it runs out of tokens. The `statement()` function looks at the current token to decide which specific statement parsing function (`printStatement()` , `ifStatement()` , etc.) to invoke.

> The critical nuance is that parsing functions *predict* what they are about to parse based on the current token, then delegate to more specific functions. This is a top-down, predictive parsing strategy.

ADR: Recursive Descent vs. Pratt Parsing

> **Decision: Use Classic Recursive Descent for Expression Parsing**
>
> - **Context**: We need a parsing algorithm for Lox expressions that correctly handles operator precedence and associativity, is understandable for learners, and maps clearly to the language grammar.
> - **Options Considered**:
>   1. **Classic Recursive Descent**: One parsing function per precedence level, with explicit calls between them forming the precedence hierarchy.
>   2. **Pratt Parsing (Top-Down Operator Precedence)**: A more compact algorithm that uses a table of precedence values and generic `parseExpression` calls, with binding power driving recursion.
>   3. **Parser Generator (e.g., ANTLR)**: Use a tool to generate a parser from a formal grammar specification.
> - **Decision**: We choose **Classic Recursive Descent**.
> - **Rationale**:
>   - **Explicitness & Educational Value**: The call graph of functions (`expression` → `equality` → `comparison` → ...) provides a direct, visual mapping to the precedence table. A learner can read the code and immediately see how precedence is implemented. Pratt parsing, while elegant, condenses this logic into a less intuitive loop and table lookup.
>   - **Simplicity for Statements**: Recursive descent handles the statement grammar (which has diverse starting keywords like `if` , `while` , `for`) very naturally with simple `if` / `else if` checks on the lookahead token. Pratt parsing is primarily designed for expressions.
>   - **Control and Error Reporting**: Having a dedicated function for each grammar rule makes it easier to craft specific, helpful error messages when a parse error occurs at a particular point (e.g., "Expect ')' after expression" inside `grouping()`).
> - **Consequences**:
>   - **More Boilerplate**: We will have ~8 functions for expression precedence levels, each with similar looping structures. This is more code than a Pratt parser.
>   - **Clear Structure**: The resulting parser is longer but extremely easy to debug and extend. Adding a new operator precedence level is as simple as adding a new function and slotting it into the call chain.
>   - **Direct Implementation**: It avoids the need to understand the somewhat subtle "binding power" concept of Pratt parsing, lowering the initial learning barrier.

Option	Pros	Cons	Chosen?
Classic Recursive Descent	Direct mapping to grammar. Easy to understand and debug. Excellent for error reporting. Handles statements naturally.	More verbose code. Requires careful ordering of function calls to get precedence right.	**Yes**
Pratt Parsing	Very concise for expressions. Elegant handling of precedence and associativity. Easy to add new operators.	More abstract, harder to understand initially. Less straightforward for non-expression grammar rules.	No
Parser Generator	Automatically handles complex grammars. Can generate efficient code.	Adds a build-time dependency. Hides the parsing logic, reducing educational value. Error messages can be cryptic.	No

Common Pitfalls

⚠️ Pitfall: Left-Recursion Infinite Loop

- **Description**: Writing a grammar rule that directly or indirectly calls itself without consuming a token first, e.g., `expression() -> expression() + term()`. In recursive descent, this leads to infinite recursion and a stack overflow.
- **Why it's wrong**: The parser function immediately calls itself again, making no progress on the token stream.
- **Fix**: Always structure grammar rules to consume at least one token (a *terminal*) before any recursive call. The standard pattern is to have a rule delegate to a higher-precedence rule first (e.g., `expression() -> equality()`, and `equality() -> comparison ( ( "!=" \| "==" ) comparison )*`).

⚠️ Pitfall: Incorrect Precedence or Associativity

- **Description**: Getting the order of operator parsing wrong. For example, parsing `+` before `*`, or making `=` left-associative. This results in an AST that evaluates to the wrong value.
- **Why it's wrong**: The tree structure no longer reflects the intended meaning of the expression. `1 + 2 * 3` might be parsed as `(1 + 2) * 3`.
- **Fix**: Meticulously follow the precedence table. Ensure the call chain in the parser respects the table order (lowest precedence calls next level). For associativity, use a loop for left-associative operators (parsing left-to-right) and recursion for right-associative ones (like `=`).

⚠️ Pitfall: Poor Error Reporting and Recovery

- **Description**: The parser throws a generic "syntax error" without indicating location or what was expected, or it crashes completely on the first error, making it hard to find multiple issues.
- **Why it's wrong**: A user cannot effectively debug their Lox code.
- **Fix**: **Synchronized panic-mode recovery**. When a parse error is detected, report the line, the token, and a context-specific message. Then, discard tokens until reaching a known "synchronization point" (like a semicolon or a statement-starting keyword) before attempting to continue parsing. This allows reporting multiple independent errors in one run.

⚠️ Pitfall: Visitor Pattern Boilerplate Overload

- **Description**: Manually writing the `accept` method in every AST node class and the visitor interface with a dozen `visit` methods is tedious and error-prone.
- **Why it's wrong**: It distracts from the core logic and introduces risk of typos or missing methods.
- **Fix**: Use your IDE's code generation features (if available) or write a small script to generate the skeleton. In the educational context, writing it once reinforces understanding of the pattern. The provided implementation guidance includes the complete boilerplate to copy.

⚠️ Pitfall: Forgetting to Handle the End-of-File (EOF) Token

- **Description**: The parser's loop or matching logic doesn't explicitly check for the `EOF` token, leading to infinite loops or `IndexOutOfBounds` exceptions when the token list is exhausted.
- **Why it's wrong**: Every valid program ends with `EOF`. The parser must gracefully stop when it reaches it.
- **Fix**: The helper method `isAtEnd()` should check if the current token is of type `EOF`. All parsing functions should use this before attempting to consume tokens.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
AST Definition	Manual Java classes with fields and Visitor interfaces.	Use an Annotation Processor or Kotlin data classes to reduce boilerplate (out of scope for learning).
Parsing Algorithm	Classic Recursive Descent as described.	Pratt Parsing for a more compact, generalized expression parser.
Error Handling	Simple `throw new ParseError(...)` with line number.	Collect multiple errors in a list before throwing, or implement more sophisticated recovery.

B. Recommended File/Module Structure

```
lox/
├── Lox.java                # Main CLI/REPL entry point
├── Token.java              # Token data class
├── TokenType.java          # TokenType enum
│
├── scanner/
│   └── Scanner.java        # Milestone 1 component
│
├── ast/                    # Milestone 2: AST Definitions
│   ├── Expr.java           # Abstract Expr base class and Visitor
│   └── Stmt.java           # Abstract Stmt base class and Visitor
│
├── parser/                 # Milestone 3: Parser
│   ├── Parser.java
│   └── ParseError.java     # Checked exception for parse errors
│
└── interpreter/            # Future milestones
    └── ... (Interpreter, Environment, etc.)
```

C. Infrastructure Starter Code

The following is the complete, ready-to-use boilerplate for the AST node classes and Visitor interfaces. This is foundational infrastructure; the learner should copy this exactly and then focus on implementing the parser logic.

`ast/Expr.java`

JAVA

```java
package com.craftinginterpreters.lox.ast;

import com.craftinginterpreters.lox.Token;
import java.util.List;

public abstract class Expr {
  public interface Visitor<R> {
    R visitBinaryExpr(Binary expr);
    R visitUnaryExpr(Unary expr);
    R visitGroupingExpr(Grouping expr);
    R visitLiteralExpr(Literal expr);
    R visitVariableExpr(Variable expr);
    R visitAssignExpr(Assign expr);
    R visitLogicalExpr(Logical expr);
    R visitCallExpr(Call expr);
    R visitGetExpr(Get expr);
    R visitSetExpr(Set expr);
    R visitThisExpr(This expr);
    R visitSuperExpr(Super expr);
  }

  public abstract <R> R accept(Visitor<R> visitor);

  public static class Binary extends Expr {
    public Binary(Expr left, Token operator, Expr right) {
      this.left = left;
      this.operator = operator;
      this.right = right;
    }

    @Override
    public <R> R accept(Visitor<R> visitor) {
      return visitor.visitBinaryExpr(this);
    }

    public final Expr left;
    public final Token operator;
    public final Expr right;
```

```
}

public static class Unary extends Expr {
  public Unary(Token operator, Expr right) {
    this.operator = operator;
    this.right = right;
  }

  @Override
  public <R> R accept(Visitor<R> visitor) {
    return visitor.visitUnaryExpr(this);
  }

  public final Token operator;
  public final Expr right;
}

public static class Grouping extends Expr {
  public Grouping(Expr expression) {
    this.expression = expression;
  }

  @Override
  public <R> R accept(Visitor<R> visitor) {
    return visitor.visitGroupingExpr(this);
  }

  public final Expr expression;
}

public static class Literal extends Expr {
  public Literal(Object value) {
    this.value = value;
  }

  @Override
  public <R> R accept(Visitor<R> visitor) {
    return visitor.visitLiteralExpr(this);
  }
```

```
    public final Object value;
  }

  public static class Variable extends Expr {
    public Variable(Token name) {
      this.name = name;
    }

    @Override
    public <R> R accept(Visitor<R> visitor) {
      return visitor.visitVariableExpr(this);
    }

    public final Token name;
  }

  public static class Assign extends Expr {
    public Assign(Token name, Expr value) {
      this.name = name;
      this.value = value;
    }

    @Override
    public <R> R accept(Visitor<R> visitor) {
      return visitor.visitAssignExpr(this);
    }

    public final Token name;
    public final Expr value;
  }

  public static class Logical extends Expr {
    public Logical(Expr left, Token operator, Expr right) {
      this.left = left;
      this.operator = operator;
      this.right = right;
    }

    @Override
```

```
    public <R> R accept(Visitor<R> visitor) {
      return visitor.visitLogicalExpr(this);
    }

    public final Expr left;
    public final Token operator;
    public final Expr right;
  }

  public static class Call extends Expr {
    public Call(Expr callee, Token paren, List<Expr> arguments) {
      this.callee = callee;
      this.paren = paren;
      this.arguments = arguments;
    }

    @Override
    public <R> R accept(Visitor<R> visitor) {
      return visitor.visitCallExpr(this);
    }

    public final Expr callee;
    public final Token paren;
    public final List<Expr> arguments;
  }

  public static class Get extends Expr {
    public Get(Expr object, Token name) {
      this.object = object;
      this.name = name;
    }

    @Override
    public <R> R accept(Visitor<R> visitor) {
      return visitor.visitGetExpr(this);
    }

    public final Expr object;
    public final Token name;
```

```
}

public static class Set extends Expr {
  public Set(Expr object, Token name, Expr value) {
    this.object = object;
    this.name = name;
    this.value = value;
  }

  @Override
  public <R> R accept(Visitor<R> visitor) {
    return visitor.visitSetExpr(this);
  }

  public final Expr object;
  public final Token name;
  public final Expr value;
}

public static class This extends Expr {
  public This(Token keyword) {
    this.keyword = keyword;
  }

  @Override
  public <R> R accept(Visitor<R> visitor) {
    return visitor.visitThisExpr(this);
  }

  public final Token keyword;
}

public static class Super extends Expr {
  public Super(Token keyword, Token method) {
    this.keyword = keyword;
    this.method = method;
  }

  @Override
```

```
    public <R> R accept(Visitor<R> visitor) {
      return visitor.visitSuperExpr(this);
    }

    public final Token keyword;
    public final Token method;
  }
}
```

ast/Stmt.java

JAVA

```java
package com.craftinginterpreters.lox.ast;

import com.craftinginterpreters.lox.Token;
import java.util.List;

public abstract class Stmt {
  public interface Visitor<V> {
    V visitExpressionStmt(Expression stmt);
    V visitPrintStmt(Print stmt);
    V visitVarStmt(Var stmt);
    V visitBlockStmt(Block stmt);
    V visitIfStmt(If stmt);
    V visitWhileStmt(While stmt);
    V visitFunctionStmt(Function stmt);
    V visitReturnStmt(Return stmt);
    V visitClassStmt(Class stmt);
  }

  public abstract <V> V accept(Visitor<V> visitor);

  public static class Expression extends Stmt {
    public Expression(Expr expression) {
      this.expression = expression;
    }

    @Override
    public <V> V accept(Visitor<V> visitor) {
      return visitor.visitExpressionStmt(this);
    }

    public final Expr expression;
  }

  public static class Print extends Stmt {
    public Print(Expr expression) {
      this.expression = expression;
    }

    @Override
```

```
    public <V> V accept(Visitor<V> visitor) {
      return visitor.visitPrintStmt(this);
    }

    public final Expr expression;
  }

  public static class Var extends Stmt {
    public Var(Token name, Expr initializer) {
      this.name = name;
      this.initializer = initializer;
    }

    @Override
    public <V> V accept(Visitor<V> visitor) {
      return visitor.visitVarStmt(this);
    }

    public final Token name;
    public final Expr initializer;
  }

  public static class Block extends Stmt {
    public Block(List<Stmt> statements) {
      this.statements = statements;
    }

    @Override
    public <V> V accept(Visitor<V> visitor) {
      return visitor.visitBlockStmt(this);
    }

    public final List<Stmt> statements;
  }

  public static class If extends Stmt {
    public If(Expr condition, Stmt thenBranch, Stmt elseBranch) {
      this.condition = condition;
      this.thenBranch = thenBranch;
```

```
      this.elseBranch = elseBranch;
    }

    @Override
    public <V> V accept(Visitor<V> visitor) {
      return visitor.visitIfStmt(this);
    }

    public final Expr condition;
    public final Stmt thenBranch;
    public final Stmt elseBranch;
  }

  public static class While extends Stmt {
    public While(Expr condition, Stmt body) {
      this.condition = condition;
      this.body = body;
    }

    @Override
    public <V> V accept(Visitor<V> visitor) {
      return visitor.visitWhileStmt(this);
    }

    public final Expr condition;
    public final Stmt body;
  }

  public static class Function extends Stmt {
    public Function(Token name, List<Token> params, List<Stmt> body) {
      this.name = name;
      this.params = params;
      this.body = body;
    }

    @Override
    public <V> V accept(Visitor<V> visitor) {
      return visitor.visitFunctionStmt(this);
    }
```

```
    public final Token name;
    public final List<Token> params;
    public final List<Stmt> body;
  }

  public static class Return extends Stmt {
    public Return(Token keyword, Expr value) {
      this.keyword = keyword;
      this.value = value;
    }

    @Override
    public <V> V accept(Visitor<V> visitor) {
      return visitor.visitReturnStmt(this);
    }

    public final Token keyword;
    public final Expr value;
  }

  public static class Class extends Stmt {
    public Class(Token name, Expr.Variable superclass, List<Stmt.Function> methods) {
      this.name = name;
      this.superclass = superclass;
      this.methods = methods;
    }

    @Override
    public <V> V accept(Visitor<V> visitor) {
      return visitor.visitClassStmt(this);
    }

    public final Token name;
    public final Expr.Variable superclass;
    public final List<Stmt.Function> methods;
  }
}
```

`parser/ParseError.java`

```java
package com.craftinginterpreters.lox.parser;


// This is a checked exception to allow simple error recovery

// in the recursive descent parser.

public class ParseError extends RuntimeException {

  // No need to store extra state; the error message is sufficient.

}
```

D. Core Logic Skeleton Code

`parser/Parser.java`

JAVA

```java
package com.craftinginterpreters.lox.parser;

import com.craftinginterpreters.lox.Token;
import com.craftinginterpreters.lox.TokenType;
import com.craftinginterpreters.lox.ast.Expr;
import com.craftinginterpreters.lox.ast.Stmt;
import java.util.ArrayList;
import java.util.List;

import static com.craftinginterpreters.lox.TokenType.*;

public class Parser {
  private final List<Token> tokens;
  private int current = 0;

  public Parser(List<Token> tokens) {
    this.tokens = tokens;
  }

  public List<Stmt> parse() {
    List<Stmt> statements = new ArrayList<>();
    while (!isAtEnd()) {
      // TODO 1: Parse a single statement and add it to the list.
      // Hint: Use declaration() which can parse any statement.
      statements.add(declaration());
    }
    return statements;
  }

  // declaration → varDecl | funDecl | classDecl | statement ;
  private Stmt declaration() {
    try {
      // TODO 2: Look at the current token to decide what kind of declaration this is.
      // If it's VAR, call varDecl().
      // If it's FUN, call funDecl("function").
      // If it's CLASS, call classDecl().
      // Otherwise, it's a statement; call statement().
      if (match(VAR)) return varDecl();
```

```
      if (match(FUN)) return funDecl("function");
      if (match(CLASS)) return classDecl();
      return statement();
    } catch (ParseError error) {
      synchronize();
      return null; // Return null to allow parsing to continue after an error.
    }
  }

  // varDecl → "var" IDENTIFIER ( "=" expression )? ";" ;
  private Stmt varDecl() {
    // TODO 3: Consume the identifier token (the variable name).
    Token name = consume(IDENTIFIER, "Expect variable name.");

    // TODO 4: Check for an initializer (an '=' token).
    Expr initializer = null;
    if (match(EQUAL)) {
      initializer = expression();
    }

    // TODO 5: Expect a semicolon after the variable declaration.
    consume(SEMICOLON, "Expect ';' after variable declaration.");
    return new Stmt.Var(name, initializer);
  }

  // funDecl → "fun" function ;
  // function → IDENTIFIER "(" parameters? ")" block ;
  // parameters → IDENTIFIER ( "," IDENTIFIER )* ;
  private Stmt funDecl(String kind) {
    // TODO 6: Consume the function name identifier.
    Token name = consume(IDENTIFIER, "Expect " + kind + " name.");

    // TODO 7: Expect a '(' for the parameter list.
    consume(LEFT_PAREN, "Expect '(' after " + kind + " name.");

    // TODO 8: Parse the parameter list.
    List<Token> parameters = new ArrayList<>();
    if (!check(RIGHT_PAREN)) {
```

```
      do {
        if (parameters.size() >= 255) {
          error(peek(), "Cannot have more than 255 parameters.");
        }
        parameters.add(consume(IDENTIFIER, "Expect parameter name."));
      } while (match(COMMA));
    }
    consume(RIGHT_PAREN, "Expect ')' after parameters.");

    // TODO 9: Parse the function body as a block.
    consume(LEFT_BRACE, "Expect '{' before " + kind + " body.");
    List<Stmt> body = block();
    return new Stmt.Function(name, parameters, body);
  }

  // classDecl → "class" IDENTIFIER ( "<" IDENTIFIER )? "{" function* "}" ;
  private Stmt classDecl() {
    // TODO 10: Consume the class name identifier.
    Token name = consume(IDENTIFIER, "Expect class name.");

    // TODO 11: Check for a superclass inheritance clause.
    Expr.Variable superclass = null;
    if (match(LESS)) {
      consume(IDENTIFIER, "Expect superclass name.");
      superclass = new Expr.Variable(previous());
    }

    // TODO 12: Expect a '{' before the class body.
    consume(LEFT_BRACE, "Expect '{' before class body.");

    // TODO 13: Parse the list of method declarations inside the class.
    List<Stmt.Function> methods = new ArrayList<>();
    while (!check(RIGHT_BRACE) && !isAtEnd()) {
      methods.add(funDecl("method"));
    }

    consume(RIGHT_BRACE, "Expect '}' after class body.");
    return new Stmt.Class(name, superclass, methods);
```

```java
}

// statement → exprStmt | printStmt | block | ifStmt | whileStmt | forStmt | returnStmt ;
private Stmt statement() {
  // TODO 14: Check the current token to decide which statement type to parse.
  if (match(PRINT)) return printStmt();
  if (match(LEFT_BRACE)) return new Stmt.Block(block());
  if (match(IF)) return ifStmt();
  if (match(WHILE)) return whileStmt();
  if (match(FOR)) return forStmt();
  if (match(RETURN)) return returnStmt();
  return expressionStmt();
}

// exprStmt → expression ";" ;
private Stmt expressionStmt() {
  // TODO 15: Parse an expression, expect a semicolon, and wrap it in an Expression statement.
  Expr expr = expression();
  consume(SEMICOLON, "Expect ';' after expression.");
  return new Stmt.Expression(expr);
}

// printStmt → "print" expression ";" ;
private Stmt printStmt() {
  // TODO 16: Parse an expression, expect a semicolon, and wrap it in a Print statement.
  Expr value = expression();
  consume(SEMICOLON, "Expect ';' after value.");
  return new Stmt.Print(value);
}

// block → "{" declaration* "}" ;
private List<Stmt> block() {
  List<Stmt> statements = new ArrayList<>();
  while (!check(RIGHT_BRACE) && !isAtEnd()) {
    statements.add(declaration());
  }
  consume(RIGHT_BRACE, "Expect '}' after block.");
```

```
    return statements;
  }

  // ifStmt → "if" "(" expression ")" statement ( "else" statement )? ;
  private Stmt ifStmt() {
    // TODO 17: Expect '(' after 'if', parse the condition, expect ')'.
    consume(LEFT_PAREN, "Expect '(' after 'if'.");
    Expr condition = expression();
    consume(RIGHT_PAREN, "Expect ')' after if condition.");

    // TODO 18: Parse the then branch statement.
    Stmt thenBranch = statement();

    // TODO 19: Check for an else token and parse the else branch if present.
    Stmt elseBranch = null;
    if (match(ELSE)) {
      elseBranch = statement();
    }
    return new Stmt.If(condition, thenBranch, elseBranch);
  }

  // whileStmt → "while" "(" expression ")" statement ;
  private Stmt whileStmt() {
    // TODO 20: Expect '(' after 'while', parse the condition, expect ')'.
    consume(LEFT_PAREN, "Expect '(' after 'while'.");
    Expr condition = expression();
    consume(RIGHT_PAREN, "Expect ')' after condition.");
    Stmt body = statement();
    return new Stmt.While(condition, body);
  }

  // forStmt → "for" "(" ( varDecl | exprStmt | ";" ) expression? ";" expression? ")" statement ;
  // This is a "desugaring" parser: it transforms for-loop syntax into an equivalent while-loop AST.
  private Stmt forStmt() {
    // TODO 21: Expect '(' after 'for'.
    consume(LEFT_PAREN, "Expect '(' after 'for'.");

    // TODO 22: Parse the initializer (could be a var declaration, an expression statement, or empty).
```

```
Stmt initializer;
if (match(SEMICOLON)) {
  initializer = null;
} else if (match(VAR)) {
  initializer = varDecl();
} else {
  initializer = expressionStmt();
}

// TODO 23: Parse the loop condition. If missing, treat it as 'true'.
Expr condition = null;
if (!check(SEMICOLON)) {
  condition = expression();
}
consume(SEMICOLON, "Expect ';' after loop condition.");

// TODO 24: Parse the increment expression. If missing, treat it as empty.
Expr increment = null;
if (!check(RIGHT_PAREN)) {
  increment = expression();
}
consume(RIGHT_PAREN, "Expect ')' after for clauses.");

// TODO 25: Parse the body statement.
Stmt body = statement();

// TODO 26: Desugar: Build the AST for a block containing initializer, a while loop with condition and body,
// and the increment executed at the end of each loop iteration.
// Hint: If increment exists, the body becomes a block: { original_body; increment; }
// Wrap that in a while loop with the condition (or true).
// If initializer exists, the whole thing becomes a block: { initializer; while_loop; }
if (increment != null) {
  body = new Stmt.Block(List.of(body, new Stmt.Expression(increment)));
}
if (condition == null) condition = new Expr.Literal(true);
body = new Stmt.While(condition, body);
if (initializer != null) {
```

```
    body = new Stmt.Block(List.of(initializer, body));
  }
  return body;
}

// returnStmt → "return" expression? ";" ;
private Stmt returnStmt() {
  Token keyword = previous();
  Expr value = null;
  if (!check(SEMICOLON)) {
    value = expression();
  }
  consume(SEMICOLON, "Expect ';' after return value.");
  return new Stmt.Return(keyword, value);
}

// expression → assignment ;
private Expr expression() {
  return assignment();
}

// assignment → ( call "." )? IDENTIFIER "=" assignment | logical_or ;
private Expr assignment() {
  // TODO 27: Parse a logical_or expression (this will parse the left-hand side).
  Expr expr = logicalOr();

  // TODO 28: Check if the next token is an '=' (assignment).
  if (match(EQUAL)) {
    Token equals = previous();
    // TODO 29: Recursively parse the right-hand side (the value being assigned).
    Expr value = assignment();

    // TODO 30: Check if the left-hand side (expr) is a valid assignment target.
    // Valid targets are: Variable, Get (property access).
    if (expr instanceof Expr.Variable) {
      Token name = ((Expr.Variable)expr).name;
      return new Expr.Assign(name, value);
    } else if (expr instanceof Expr.Get) {
```

```
      Expr.Get get = (Expr.Get)expr;
      return new Expr.Set(get.object, get.name, value);
    }
    // If not, report an error (invalid assignment target).
    error(equals, "Invalid assignment target.");
  }
  return expr;
}

// logical_or → logical_and ( "or" logical_and )* ;
private Expr logicalOr() {
  // TODO 31: Parse a logical_and expression.
  Expr expr = logicalAnd();

  // TODO 32: While the next token is 'or', consume it and parse the right-hand logical_and.
  while (match(OR)) {
    Token operator = previous();
    Expr right = logicalAnd();
    expr = new Expr.Logical(expr, operator, right);
  }
  return expr;
}

// logical_and → equality ( "and" equality )* ;
private Expr logicalAnd() {
  // TODO 33: Parse an equality expression.
  Expr expr = equality();

  // TODO 34: While the next token is 'and', consume it and parse the right-hand equality.
  while (match(AND)) {
    Token operator = previous();
    Expr right = equality();
    expr = new Expr.Logical(expr, operator, right);
  }
  return expr;
}

// equality → comparison ( ( "!=" | "==" ) comparison )* ;
```

```
private Expr equality() {
  // TODO 35: Parse a comparison expression.
  Expr expr = comparison();

  // TODO 36: While the next token is '!=' or '==', consume it and parse the next comparison.
  while (match(BANG_EQUAL, EQUAL_EQUAL)) {
    Token operator = previous();
    Expr right = comparison();
    expr = new Expr.Binary(expr, operator, right);
  }
  return expr;
}

// comparison → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
private Expr comparison() {
  // TODO 37: Parse a term expression.
  Expr expr = term();

  // TODO 38: While the next token is '>', '>=', '<', or '<=', consume it and parse the next term.
  while (match(GREATER, GREATER_EQUAL, LESS, LESS_EQUAL)) {
    Token operator = previous();
    Expr right = term();
    expr = new Expr.Binary(expr, operator, right);
  }
  return expr;
}

// term → factor ( ( "-" | "+" ) factor )* ;
private Expr term() {
  // TODO 39: Parse a factor expression.
  Expr expr = factor();

  // TODO 40: While the next token is '-' or '+', consume it and parse the next factor.
  while (match(MINUS, PLUS)) {
    Token operator = previous();
    Expr right = factor();
    expr = new Expr.Binary(expr, operator, right);
  }
```

```
    return expr;
  }

  // factor → unary ( ( "/" | "*" ) unary )* ;
  private Expr factor() {
    // TODO 41: Parse a unary expression.
    Expr expr = unary();

    // TODO 42: While the next token is '/' or '*', consume it and parse the next unary.
    while (match(SLASH, STAR)) {
      Token operator = previous();
      Expr right = unary();
      expr = new Expr.Binary(expr, operator, right);
    }
    return expr;
  }

  // unary → ( "!" | "-" ) unary | call ;
  private Expr unary() {
    // TODO 43: If the next token is '!' or '-', consume it, recursively parse a unary, and return a Unary node.
    if (match(BANG, MINUS)) {
      Token operator = previous();
      Expr right = unary();
      return new Expr.Unary(operator, right);
    }
    // TODO 44: Otherwise, parse a call expression.
    return call();
  }

  // call → primary ( "(" arguments? ")" | "." IDENTIFIER )* ;
  private Expr call() {
    // TODO 45: Start by parsing a primary expression (the leftmost part of the call chain).
    Expr expr = primary();

    while (true) {
      // TODO 46: If we see a '(', this is a function call.
      if (match(LEFT_PAREN)) {
        expr = finishCall(expr);
```

```
    } else if (match(DOT)) {
      // TODO 47: If we see a '.', this is a property access.
      Token name = consume(IDENTIFIER, "Expect property name after '.'.");
      expr = new Expr.Get(expr, name);
    } else {
      break;
    }
  }
  return expr;
}

// arguments → expression ( "," expression )* ;
private Expr finishCall(Expr callee) {
  List<Expr> arguments = new ArrayList<>();
  if (!check(RIGHT_PAREN)) {
    do {
      if (arguments.size() >= 255) {
        error(peek(), "Cannot have more than 255 arguments.");
      }
      arguments.add(expression());
    } while (match(COMMA));
  }
  Token paren = consume(RIGHT_PAREN, "Expect ')' after arguments.");
  return new Expr.Call(callee, paren, arguments);
}

// primary → NUMBER | STRING | "true" | "false" | "nil" | "this"
//         | "(" expression ")" | IDENTIFIER | "super" "." IDENTIFIER ;
private Expr primary() {
  // TODO 48: Handle literal tokens: NUMBER, STRING, TRUE, FALSE, NIL.
  if (match(FALSE)) return new Expr.Literal(false);
  if (match(TRUE)) return new Expr.Literal(true);
  if (match(NIL)) return new Expr.Literal(null);
  if (match(NUMBER, STRING)) {
    return new Expr.Literal(previous().literal);
  }
```

```java
    // TODO 49: Handle the 'this' keyword.
    if (match(THIS)) return new Expr.This(previous());

    // TODO 50: Handle the 'super' keyword for superclass method calls.
    if (match(SUPER)) {
      Token keyword = previous();
      consume(DOT, "Expect '.' after 'super'.");
      Token method = consume(IDENTIFIER, "Expect superclass method name.");
      return new Expr.Super(keyword, method);
    }

    // TODO 51: Handle identifier (variable name).
    if (match(IDENTIFIER)) {
      return new Expr.Variable(previous());
    }

    // TODO 52: Handle grouping parentheses.
    if (match(LEFT_PAREN)) {
      Expr expr = expression();
      consume(RIGHT_PAREN, "Expect ')' after expression.");
      return new Expr.Grouping(expr);
    }

    // TODO 53: If none of the above match, throw a parse error.
    throw error(peek(), "Expect expression.");
  }

  // ========== Parser Utility Methods ==========
  private boolean match(TokenType... types) {
    for (TokenType type : types) {
      if (check(type)) {
        advance();
        return true;
      }
    }
    return false;
  }
```

```
private Token consume(TokenType type, String message) {
  if (check(type)) return advance();
  throw error(peek(), message);
}

private boolean check(TokenType type) {
  if (isAtEnd()) return false;
  return peek().type == type;
}

private Token advance() {
  if (!isAtEnd()) current++;
  return previous();
}

private boolean isAtEnd() {
  return peek().type == EOF;
}

private Token peek() {
  return tokens.get(current);
}

private Token previous() {
  return tokens.get(current - 1);
}

private ParseError error(Token token, String message) {
  Lox.error(token, message); // Assume Lox.error reports the error to the user.
  return new ParseError();
}

private void synchronize() {
  advance();
  while (!isAtEnd()) {
    if (previous().type == SEMICOLON) return;
    switch (peek().type) {
      case CLASS:
      case FUN:
```

```
        case VAR:
        case FOR:
        case IF:
        case WHILE:
        case PRINT:
        case RETURN:
          return;
      }
      advance();
    }
  }
}
```

E. Language-Specific Hints

- Use `final` fields in AST node classes to enforce immutability.
- The `Object` type for `Literal.value` and `Token.literal` can hold `Double`, `String`, `Boolean`, or `null` (for `nil`). Be cautious with autoboxing.
- Use `List<Stmt>` and `List<Expr>` from `java.util`. For mutable lists during parsing, use `ArrayList`.
- The `ParseError` class extends `RuntimeException` but is used as a checked exception for control flow within the parser's error recovery. This is a slight abuse but keeps code simple.
- The `synchronize()` method uses a `switch` on `TokenType`; ensure your `TokenType` enum includes all statement-starting keywords.

F. Milestone Checkpoint After implementing the AST classes and the parser (Milestones 2 & 3), you can verify your work with a simple test:

1. **Create a test Lox program**, `test.lox`:

```
var a = 1 + 2 * 3;
print a;
if (a > 5) {
  print "greater";
} else {
  print "less or equal";
}
```

2. **Add a temporary main method or test** that reads this file, runs the scanner to get tokens, passes them to the parser, and prints the resulting AST using a `PrettyPrinter` visitor (which you should implement as part of Milestone 2).
3. **Run your test**. The expected output should be a correctly parenthesized S-expression reflecting precedence and grouping:

```
(block
  (var a (= (group (+ 1 (* 2 3)))))
  (print (variable a))
  (if (> (variable a) 5)
    (block (print "greater"))
    (block (print "less or equal")))
)
```

4. **Signs of Success**: The parser runs without throwing a `ParseError`. The AST structure matches the expected precedence (`*` before `+`). The `if` statement's condition and branches are correctly nested.
5. **Common Failure Modes**:

- **Infinite loop/stack overflow**: Likely due to left-recursion in a grammar rule. Review your `expression()`, `equality()`, etc., to ensure they always consume a token before any recursive call.
- **Incorrect AST (wrong precedence)**: Verify the order of calls in your expression parsing cascade. The function for lower precedence (like `term`) must call the function for higher precedence (`factor`), not the other way around.
- **"Expect expression" error on valid code**: Your `primary()` method might not handle all token types (like `IDENTIFIER` or `NUMBER`). Ensure all literal types and the grouping `(` are covered.

> **Milestone(s):** Milestone 4 - Evaluating Expressions, Milestone 5 - Statements and State, Milestone 6 - Control Flow

5.3 Component Design: Core Interpreter [Milestones 4, 5, 6]

This section details the design of the **Interpreter**, the component responsible for bringing Lox programs to life. It takes the static Abstract Syntax Tree produced by the parser and animates it by walking its structure, evaluating expressions and executing statements to produce observable side effects. This component introduces dynamic runtime behavior, state management through environments, and control flow—transforming a declarative program description into executable logic.

Mental Model: The Tree-Walking Executor

Imagine the interpreter as a puppeteer, and the AST as a complex marionette. The puppeteer doesn't see the entire puppet at once but instead walks along its strings (the AST edges), visiting each joint (node) in a predetermined order. At each joint, the puppeteer performs a specific action based on the joint's type: for a **`Binary`** node, they calculate a mathematical result by combining the values from its two child strings; for an **`If`** node, they check a condition and decide which branch string to follow next. The puppeteer carries with them a notebook—the **`Environment`**—where they record and look up variable names and their current values as they move through different scopes (like changing rooms in a theater). This hands-on, step-by-step navigation of the tree structure is the essence of tree-walking interpretation—direct, intuitive, and a perfect match for educational implementation.

Interface: The Evaluate and Execute Methods

The interpreter's public interface is minimal, centered on two core dispatch methods that serve as entry points for interpreting different types of AST nodes. Internally, it maintains the runtime state through an environment chain and handles control flow.

Method	Parameters	Returns	Description
`interpret`	`List<Stmt>` statements	`void`	The primary public method. Takes a list of statements (the top-level program) and executes them sequentially. Internally, it calls `execute` on each statement.
`evaluate`	`Expr` expression	`Object` (runtime value)	Dispatches to the appropriate `visit` method for an expression node, recursively evaluates its subexpressions, and returns the computed runtime value (e.g., a Java `Double` for numbers, `String` for strings).
`execute`	`Stmt` statement	`void`	Dispatches to the appropriate `visit` method for a statement node, performing its side effects (e.g., printing, variable assignment, control flow).
`executeBlock`	`List<Stmt>` statements, `Environment` environment	`void`	A specialized method for executing a block of statements within a new, nested environment. This is called from `visitBlockStmt` and is key to implementing lexical scoping.

The interpreter also maintains critical internal state:

State Component	Type	Description
`globals`	`Environment`	The outermost environment, shared across all interpretations. Contains built-in functions and global variable definitions. Persists for the lifetime of the interpreter.
`environment`	`Environment`	A reference to the *current* active environment. This reference changes as the interpreter enters and exits blocks, functions, and other scopes. Initially points to `globals`.
`locals`	`Map<Expr, Integer>`	An optional optimization mapping from expression nodes (particularly `Variable` and `Assign`) to the number of environment "hops" required to resolve them. Used for efficient variable lookup in the presence of closures (detailed in Section 5.4).

The interpreter implements the `Expr.Visitor<Object>` and `Stmt.Visitor<Void>` interfaces generated in Section 5.2. Each `visit` method (e.g., `visitBinaryExpr`, `visitPrintStmt`) contains the logic to evaluate or execute that specific node type. The `evaluate` and `execute` methods are thin wrappers that call `expression.accept(this)` or `statement.accept(this)`, leveraging the Visitor pattern's double dispatch.

Evaluation and Execution Algorithms

The interpreter's logic is defined by the collection of `visit` methods. Below are detailed, step-by-step algorithms for the core expression and statement types covered in Milestones 4-6. Each algorithm assumes it is operating within the context of the current `environment`.

Expression Evaluation Algorithms

1. **`visitLiteralExpr` (for `Literal` nodes):**
    1. Return the `value` field stored in the node directly. This value is the Java object (e.g., `Double`, `String`, `Boolean`, `null` for nil) placed there by the parser after reading the token's literal.
2. **`visitGroupingExpr` (for `Grouping` nodes):**
    1. Recursively `evaluate` the inner `expression` field.
    2. Return the resulting value. The grouping parentheses only affect parsing precedence; they have no runtime effect.
3. **`visitUnaryExpr` (for `Unary` nodes):**
    1. Evaluate the `right` operand expression by calling `evaluate`.
    2. Inspect the `operator.type`:
        - If `BANG` (`!`): Apply Lox's **truthiness** rule. Determine if the operand value is "falsy" (only `false` or `nil`). Return the logical negation: `true` if the operand is falsy, `false` otherwise.
        - If `MINUS` (`-`): Check that the operand is a number (Java `Double`). If not, throw a `RuntimeError`. If it is, negate its numeric value and return the new `Double`.
    3. Return the computed result.
4. **`visitBinaryExpr` (for `Binary` nodes):**
    1. Evaluate the `left` operand expression.
    2. **Short-circuit evaluation for `OR` and `AND` (logical operators):**
        - For `OR`: If the left operand is *truthy*, return it immediately without evaluating the right operand.
        - For `AND`: If the left operand is *falsy*, return it immediately without evaluating the right operand.
    3. If not short-circuited, evaluate the `right` operand.
    4. Based on the `operator.type`, perform the operation:
        - **Arithmetic (`+`, `-`, `*`, `/`)**: Ensure both operands are numbers (`Double`). For `+`, also allow the special case where *either* operand is a `String`, in which case perform string concatenation (converting the other operand to its string representation). Perform the arithmetic operation and return the numeric result.
        - **Comparison (`>`, `>=`, `<`, `<=`)**: Ensure both operands are numbers. Perform the numeric comparison and return a Java `Boolean`.

- **Equality (`==` , `!=`)**: These operators work on operands of *any* type. Two values are equal if they are the same Java object (for references) or have the same primitive value. `nil` is only equal to `nil`.

5. If any type check fails, throw a `RuntimeError`.

5. **`visitVariableExpr` (for `Variable` nodes):**
    1. Look up the variable's name (`name.lexeme`) in the environment chain. Use `environment.get(name)`.
    2. If the name is found, return its bound value.
    3. If the name is not found (i.e., the variable is undefined), throw a `RuntimeError`.
6. **`visitAssignExpr` (for `Assign` nodes):**
    1. Evaluate the `value` expression to get the new value.
    2. Assign this value to the variable named `name.lexeme` in the appropriate environment. Use `environment.assign(name, value)`. This method searches outward through the environment chain for an existing binding and updates it.
    3. If the variable is not found (was never declared), `assign` throws a `RuntimeError`.
    4. Return the assigned value (assignment is an expression in Lox).

Statement Execution Algorithms

1. **`visitExpressionStmt` (for `Expression` nodes):**
    1. Evaluate the contained `expression`.
    2. Discard the result. This statement exists solely for side effects (e.g., a function call or assignment).
2. **`visitPrintStmt` (for `Print` nodes):**
    1. Evaluate the `expression`.
    2. Convert the resulting value to its string representation (using a helper `stringify` method that handles `nil` and numbers without trailing `.0`).
    3. Print the string to standard output, typically followed by a newline.
3. **`visitVarStmt` (for `Var` nodes):**
    1. Initialize a value: if the `initializer` field is not `null`, evaluate it; otherwise, use `nil` (`null` in Java).
    2. Define a new variable in the *current* environment with `name.lexeme` bound to this value. Use `environment.define(name.lexeme, value)`.
4. **`visitBlockStmt` (for `Block` nodes):**
    1. Create a new `Environment` object whose enclosing/parent environment is the *current* `environment`.
    2. Execute the list of `statements` within this newly created environment. This is done by calling the dedicated `executeBlock` method, which temporarily swaps the interpreter's `environment` reference, runs the statements, and then restores the previous environment.
    3. This nested environment is discarded when the block finishes, implementing block-level scoping.
5. **`visitIfStmt` (for `If` nodes):**
    1. Evaluate the `condition` expression.
    2. Apply the truthiness rule: if the result is truthy (not `false` or `nil`), `execute` the `thenBranch` statement.
    3. Otherwise, if an `elseBranch` exists, `execute` it.
6. **`visitWhileStmt` (for `While` nodes):**
    1. Evaluate the `condition`.
    2. If the condition is truthy:
        - `Execute` the `body` statement.
        - Jump back to step 1 (re-evaluate the condition).
    3. If the condition is falsy, exit the loop and continue with the next statement.
7. **`visitLogicalExpr` (for `Logical` nodes):** (While an expression, its algorithm is closely tied to control flow via short-circuiting, as described in step 4.ii of the binary expression algorithm above).

ADR: Dynamic Type Checking Strategy

Decision: Perform Runtime Type Checks at Each Operation Site

- **Context**: Lox is a dynamically typed language. The type of an expression (number, string, boolean, etc.) is only known at runtime when the expression is evaluated. We must ensure operations like addition or comparison are only applied to compatible operand types to maintain language semantics and prevent undefined behavior.
- **Options Considered**:
  1. **Inline Type Guards**: Within each `visit` method (e.g., `visitBinaryExpr`), check the types of the evaluated operands immediately before performing the operation. Throw a `RuntimeError` on mismatch.
  2. **Centralized Type Validation**: Create a separate `TypeChecker` component that performs all type validation in a separate, pre-evaluation pass over the AST. This pass would annotate nodes with expected types or flag type errors statically.
  3. **Coercive Typing**: Automatically convert operands to a common type (e.g., converting numbers to strings in `+` operations, or treating any non-number as `0` in arithmetic). This is common in languages like JavaScript but not Lox's specified semantics.
- **Decision**: We chose **Option 1: Inline Type Guards**.
- **Rationale**:
  - **Simplicity and Direct Mapping**: It is the most straightforward approach for a tree-walking interpreter. The logic for an operation (e.g., "add two numbers") and the check for its preconditions ("are both operands numbers?") are colocated, making the code easier to understand and debug for learners.
  - **Faithfulness to Lox Semantics**: Lox's specification calls for runtime type errors, not static type warnings. A separate type-checking pass (Option 2) adds significant complexity (a fourth phase to the interpreter) for a feature not required by the language.
  - **Explicit Error Messages**: Inline checks allow for very specific, context-rich error messages (e.g., "Operands must be two numbers or two strings for addition.") at the exact point of failure.
  - **Performance Adequacy**: For an educational interpreter, the performance overhead of repeated type checks is negligible and does not justify the complexity of the other options.
- **Consequences**:
  - **Positive**: The implementation is clear, modular, and easy to extend. Adding a new operation only requires modifying one `visit` method.
  - **Negative**: Type errors are only caught when the erroneous code path is executed, not during a "compile" phase. A program with a latent type error in an unused function will run without complaint until that function is called.
  - **Trade-off Accepted**: We prioritize implementation clarity and alignment with Lox's dynamic nature over early error detection.

Option	Pros	Cons	Chosen?
Inline Type Guards	Simple, direct, clear error messages, faithful to dynamic typing.	Errors are runtime-only, not static.	**Yes**
Centralized Type Validation	Catches errors before execution, can enable optimizations.	High complexity, not required by Lox, conflates static and dynamic analysis.	No
Coercive Typing	More forgiving for programmers, can reduce runtime errors.	Changes language semantics, can hide bugs, less predictable behavior.	No

Common Pitfalls

⚠️ Pitfall: Confusing Java `null` with Lox `nil`

- **Description**: Using Java's `null` reference directly to represent Lox's `nil` can lead to `NullPointerException`s if you forget to handle `null` before calling methods like `toString()` or before performing operations. Also, the truthiness rule must explicitly treat Java `null` as falsy.
- **Why it's wrong**: It breaks the interpreter when `nil` values flow into operations or print statements. The interpreter must treat `nil` as a valid, first-class runtime value.

- **How to fix**: Create a dedicated singleton object (e.g., `public static final Object NIL = new Object();`) to represent Lox's `nil`. Alternatively, consistently check for Java `null` before any operation and treat it according to Lox's rules. A `stringify()` helper method should explicitly return the string `"nil"` for `null`.

⚠️ Pitfall: Improper Scoping Leading to Shadowing Bugs

- **Description**: Incorrectly managing the `environment` reference when entering/exiting blocks can cause variables from an outer scope to become inaccessible (if the new environment doesn't link to the parent) or cause changes in an inner scope to incorrectly modify an outer variable (if you reuse the same environment object).
- **Why it's wrong**: This violates lexical scoping rules. Variables should be correctly shadowed, and inner blocks should not pollute outer scopes unless using `var` at the outer level.
- **How to fix**: Meticulously follow the `executeBlock` pattern: 1) Create a *new* `Environment` with the current one as its parent. 2) Use a temporary variable to save the old `environment` reference. 3) Set the interpreter's `environment` to the new one. 4) Execute the statements. 5) **Restore** the saved environment reference. This ensures a clean scope hierarchy.

⚠️ Pitfall: Forgetting Short-Circuit Evaluation in Logical Operators

- **Description**: Implementing `visitLogicalExpr` by evaluating both the `left` and `right` operands unconditionally before checking the operator.
- **Why it's wrong**: This breaks the defined semantics of `and` and `or`. For `false and someFunction()`, `someFunction()` should never be called. This is not just an optimization; it's a semantic requirement that affects programs with side effects.
- **How to fix**: Implement the algorithm described in Step 4.ii of the Binary Expression algorithm. Evaluate the left operand first. For `or`, if truthy, return it. For `and`, if falsy, return it. Only evaluate the right operand if needed.

⚠️ Pitfall: Infinite Loops Without a Safety Net

- **Description**: A simple `while (true) {}` loop will cause the interpreter to hang forever, as there's no mechanism to interrupt execution.
- **Why it's wrong**: While this is semantically correct for Lox, it's a poor user experience in an educational environment, especially in a REPL.
- **How to fix**: Consider adding an optional loop iteration limit or a timeout mechanism, particularly in the REPL mode. This can be implemented as a counter in the `visitWhileStmt` method that increments each iteration and throws a `RuntimeError` if a threshold (e.g., 10,000) is exceeded. Document this as a non-standard, educational extension.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Recommended for Learning)	Advanced Option (For Further Exploration)
Value Representation	Java `Object` (using `Double`, `String`, `Boolean`, and `null`/a sentinel for `nil`)	Define a sealed hierarchy `LoxValue` with subclasses `LoxNumber`, `LoxString`, etc.
Environment Lookup	Linear chain search via `Map<String, Object>` and parent reference.	Implement persistent hash maps or indexed environments for faster lookups in deep chains.
Error Reporting	Basic `RuntimeError` with message and token line number.	Collect stack traces on error, showing the call chain.

B. Recommended File/Module Structure

```
src/
└── com/
    └── craftinginterpreters/
        └── lox/
            ├── Lox.java                # Main CLI/REPL launcher
            ├── Token.java              # Token data class
            ├── TokenType.java          # Token type enum
            ├── Scanner.java            # Lexer (Milestone 1)
            ├── Expr.java               # Expression AST nodes (Visitor pattern)
            ├── Stmt.java               # Statement AST nodes (Visitor pattern)
            ├── Parser.java             # Recursive descent parser (Milestones 2 & 3)
            ├── Interpreter.java        # **This component** (Milestones 4, 5, 6)
            ├── Environment.java        # Runtime environment chain
            ├── RuntimeError.java       # Exception for runtime errors
            └── (Later: LoxFunction.java, LoxClass.java, etc.)
```

C. Infrastructure Starter Code (COMPLETE)

RuntimeError.java (Complete):

JAVA

```java
package com.craftinginterpreters.lox;


public class RuntimeError extends RuntimeException {

    final Token token; // The token where the error occurred (for line number)


    public RuntimeError(Token token, String message) {

        super(message);

        this.token = token;

    }

}
```

Environment.java (Complete - core logic):

JAVA

```java
package com.craftinginterpreters.lox;

import java.util.HashMap;
import java.util.Map;

public class Environment {
    final Environment enclosing;
    private final Map<String, Object> values = new HashMap<>();

    // Constructor for global scope
    public Environment() {
        enclosing = null;
    }

    // Constructor for local scope
    public Environment(Environment enclosing) {
        this.enclosing = enclosing;
    }

    // Define a new variable in the current scope
    public void define(String name, Object value) {
        values.put(name, value);
    }

    // Get the value of a variable, searching outward
    public Object get(Token name) {
        if (values.containsKey(name.lexeme)) {
            return values.get(name.lexeme);
        }
        // Look in parent scope
        if (enclosing != null) return enclosing.get(name);
        // Not found anywhere
        throw new RuntimeError(name, "Undefined variable '" + name.lexeme + "'.");
    }

    // Assign to an existing variable, searching outward
    public void assign(Token name, Object value) {
        if (values.containsKey(name.lexeme)) {
            values.put(name.lexeme, value);
```

```
            return;
        }
        // Look in parent scope
        if (enclosing != null) {
            enclosing.assign(name, value);
            return;
        }
        throw new RuntimeError(name, "Undefined variable '" + name.lexeme + "'.");
    }
}
```

D. Core Logic Skeleton Code

Interpreter.java (Skeleton with TODOs):

JAVA

```java
package com.craftinginterpreters.lox;

import java.util.List;

public class Interpreter implements Expr.Visitor<Object>, Stmt.Visitor<Void> {
    // The global environment, shared across interpretations
    final Environment globals = new Environment();
    // The current active environment (changes with scope)
    private Environment environment = globals;

    public Interpreter() {
        // TODO 1 (Optional): Add native functions to globals here later
        // e.g., define a "clock()" function.
    }

    // Main public API
    public void interpret(List<Stmt> statements) {
        try {
            for (Stmt statement : statements) {
                execute(statement);
            }
        } catch (RuntimeError error) {
            Lox.runtimeError(error);
        }
    }

    // --- Expression Visitor Methods ---
    @Override
    public Object visitLiteralExpr(Expr.Literal expr) {
        // TODO 2: Return the literal value stored in the node.
        return null;
    }

    @Override
    public Object visitGroupingExpr(Expr.Grouping expr) {
        // TODO 3: Evaluate the inner expression and return its value.
        return null;
    }
```

```java
@Override
public Object visitUnaryExpr(Expr.Unary expr) {
    Object right = evaluate(expr.right);

    switch (expr.operator.type) {
        case BANG:
            // TODO 4: Return the logical negation of 'right'.
            // Apply Lox truthiness: false and nil are false, everything else true.
            return null;
        case MINUS:
            // TODO 5: Check that 'right' is a number (Double).
            // If not, throw a RuntimeError with expr.operator.
            // If it is, negate it and return the new Double.
            return null;
    }

    // Unreachable
    return null;
}

@Override
public Object visitBinaryExpr(Expr.Binary expr) {
    Object left = evaluate(expr.left);
    Object right = evaluate(expr.right); // **CAUTION**: See TODO 6 about short-circuit.

    switch (expr.operator.type) {
        case GREATER:
            // TODO 7: Check both operands are numbers. Compare, return Boolean.
            break;
        case GREATER_EQUAL:
            // TODO 8: Check both operands are numbers. Compare, return Boolean.
            break;
        case LESS:
            // TODO 9: Check both operads are numbers. Compare, return Boolean.
            break;
        case LESS_EQUAL:
            // TODO 10: Check both operands are numbers. Compare, return Boolean.
```

```
            break;
        case MINUS:
            // TODO 11: Check both operands are numbers. Subtract, return Double.
            break;
        case SLASH:
            // TODO 12: Check both operands are numbers. Divide, return Double.
            // Bonus: Check for division by zero.
            break;
        case STAR:
            // TODO 13: Check both operands are numbers. Multiply, return Double.
            break;
        case PLUS:
            // TODO 14: Handle addition and string concatenation.
            // If both are Doubles, add.
            // If both are Strings, concatenate.
            // If one is String and one is Number, convert Number to string and concatenate.
            // Otherwise, throw RuntimeError.
            break;
        case BANG_EQUAL:
            // TODO 15: Return !isEqual(left, right)
            break;
        case EQUAL_EQUAL:
            // TODO 16: Return isEqual(left, right)
            break;
        // Logical operators (short-circuit handled separately in visitLogicalExpr)
    }

    // Unreachable
    return null;
}

@Override
public Object visitVariableExpr(Expr.Variable expr) {
    // TODO 17: Look up the variable name in the environment and return its value.
    return null;
}
```

```java
    @Override
    public Object visitAssignExpr(Expr.Assign expr) {
        Object value = evaluate(expr.value);
        // TODO 18: Assign the value to the variable in the environment.
        // Use environment.assign(...). Then return the value.
        return null;
    }

    @Override
    public Object visitLogicalExpr(Expr.Logical expr) {
        Object left = evaluate(expr.left);

        // TODO 19: Implement short-circuit logic.
        // If operator is OR and left is truthy, return left.
        // If operator is AND and left is falsy, return left.
        // Otherwise, evaluate the right operand and return it.
        return null;
    }

    // --- Statement Visitor Methods ---
    @Override
    public Void visitExpressionStmt(Stmt.Expression stmt) {
        evaluate(stmt.expression);
        // TODO 20: Evaluate the expression and discard the result.
        return null;
    }

    @Override
    public Void visitPrintStmt(Stmt.Print stmt) {
        Object value = evaluate(stmt.expression);
        // TODO 21: Convert value to string using stringify() and print it.
        System.out.println(stringify(value));
        return null;
    }

    @Override
    public Void visitVarStmt(Stmt.Var stmt) {
        Object value = null;
```

```
        if (stmt.initializer != null) {
            value = evaluate(stmt.initializer);
        }
        // TODO 22: Define the variable in the current environment with this value.
        environment.define(stmt.name.lexeme, value);
        return null;
    }

    @Override
    public Void visitBlockStmt(Stmt.Block stmt) {
        // TODO 23: Execute the block of statements in a new nested environment.
        // 1. Create new Environment with current environment as parent.
        // 2. Call executeBlock(stmt.statements, newEnvironment).
        executeBlock(stmt.statements, new Environment(environment));
        return null;
    }

    @Override
    public Void visitIfStmt(Stmt.If stmt) {
        // TODO 24: Evaluate condition. If truthy, execute thenBranch.
        // If falsy and elseBranch exists, execute elseBranch.
        if (isTruthy(evaluate(stmt.condition))) {
            execute(stmt.thenBranch);
        } else if (stmt.elseBranch != null) {
            execute(stmt.elseBranch);
        }
        return null;
    }

    @Override
    public Void visitWhileStmt(Stmt.While stmt) {
        // TODO 25: While the condition evaluates to truthy, execute the body.
        while (isTruthy(evaluate(stmt.condition))) {
            execute(stmt.body);
        }
        return null;
    }
```

```java
// --- Helper Methods ---
private void execute(Stmt stmt) {
    stmt.accept(this);
}

private Object evaluate(Expr expr) {
    return expr.accept(this);
}

public void executeBlock(List<Stmt> statements, Environment environment) {
    Environment previous = this.environment;
    try {
        this.environment = environment;
        for (Stmt statement : statements) {
            execute(statement);
        }
    } finally {
        // TODO 26: Restore the previous environment, even if an exception is thrown.
        this.environment = previous;
    }
}

// Helper: Lox truthiness rule
private boolean isTruthy(Object object) {
    // TODO 27: Return false if object is null (nil) or Boolean.FALSE.
    // Return true for everything else.
    if (object == null) return false;
    if (object instanceof Boolean) return (boolean) object;
    return true;
}

// Helper: Deep equality for Lox values
private boolean isEqual(Object a, Object b) {
    // TODO 28: Handle nil/null equality.
    // If both are null, return true.
    // If only one is null, return false.
    // Otherwise, use Java's equals().
```

```
        if (a == null && b == null) return true;
        if (a == null) return false;
        return a.equals(b);
    }

    // Helper: Convert a Lox value to its string representation
    private String stringify(Object object) {
        // TODO 29: Convert object to a readable string.
        // If null, return "nil".
        // If it's a Double, remove trailing ".0" if it's an integer.
        // Otherwise, use Java's toString().
        if (object == null) return "nil";
        if (object instanceof Double) {
            String text = object.toString();
            if (text.endsWith(".0")) {
                text = text.substring(0, text.length() - 2);
            }
            return text;
        }
        return object.toString();
    }
}
```

E. Language-Specific Hints

- Use `instanceof` for type checks (e.g., `right instanceof Double`).
- Cast carefully after `instanceof` checks (e.g., `double rightValue = (Double)right;`).
- Remember that Java's `==` for `Double` objects checks reference equality, not numeric equality. Use `.equals()` or compare the primitive `doubleValue()`.
- The `isTruthy` helper is critical and used frequently. Keep its logic simple and in one place.

F. Milestone Checkpoint After implementing this section, you should be able to run Lox programs with variables, arithmetic, print statements, and control flow.

1. **Test Command**: Create a test file `test.lox`:

```lox
var a = 5;
var b = 10;
print a + b; // Should print 15

if (a < b) {
    print "a is smaller";
} else {
    print "a is not smaller";
}

var counter = 0;
while (counter < 3) {
    print counter;
    counter = counter + 1;
}
```

2. **Expected Output**:

```
15
a is smaller
0
1
2
```

3. **Verification**: Run your interpreter (e.g., `java com.craftinginterpreters.lox.Lox test.lox`). The output should match exactly.
4. **Signs of Trouble**:
   - `NullPointerException`: Likely missing null/nil handling in `stringify` or `isTruthy`.
   - `"Undefined variable"` error for a declared variable: Check that `environment.define` is being called and that scopes (`executeBlock`) are managing the environment chain correctly.
   - Infinite loop: Verify the condition in `visitWhileStmt` is being re-evaluated each iteration.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
All numbers print with ".0" (e.g., "15.0")	The `stringify` method isn't stripping the `.0` from integer-valued doubles.	Check the `stringify` logic for `Double` objects.	Implement the substring logic shown in TODO 29.
`if` statement never executes the `else` branch	The truthiness condition (`isTruthy`) is incorrect, possibly treating `false` as truthy.	Add a debug print in `isTruthy` or test with `if (false) print "wrong"; else print "right";`	Ensure `isTruthy` returns `false` for `Boolean.FALSE`.
Changing a variable inside a block affects an outer variable with the same name	The block is not creating a new environment; you're using the same `Map`.	Print the environment reference before/after entering a block.	Ensure `new Environment(environment)` is called in `visitBlockStmt`.
`while` loop runs one extra time or not at all	The condition is evaluated at the wrong time (e.g., only once before the loop).	Trace the order of `evaluate(condition)` and `execute(body)` in `visitWhileStmt`.	The condition must be evaluated *before each* iteration, including the first.

5.4 Component Design: Functions and Closures [Milestones 7 & 8]

> **Milestone(s):** Milestone 7 - Functions, Milestone 8 - Closures

This section extends the interpreter's capabilities from simple statements and expressions to **first-class functions**, **function calls**, **return statements**, and **lexical scoping via closures**. These features transform the interpreter from a calculator into a true programming language,

enabling code organization, abstraction, and powerful functional patterns.

Mental Model: The Function as a Recipe with a Kitchen

Before diving into technical details, let's build intuition with an analogy. Think of a function as a **recipe** with two essential parts:

1. **The Recipe Card (Function Declaration):** This contains the list of ingredients (parameters) and the step-by-step instructions (body statements).
2. **The Kitchen (Environment):** This is where the cooking happens, containing all the available ingredients (variables) at the time the recipe is written.

When you define a function, you're creating a recipe card. When you call it, you:

- Gather the actual ingredients (evaluate arguments)
- Set up a clean work area (create a new environment)
- Place the measured ingredients on the counter (bind parameters)
- Follow the instructions step by step (execute the body)

A **closure** is a recipe card that comes with a **snapshot of the kitchen** where it was written. Even if you later take the recipe to a different kitchen (call it from a different scope), it still has access to all the original kitchen's ingredients (variables from the defining environment). This allows inner functions to "remember" and access variables from outer functions, even after those outer functions have finished executing.

> The critical insight: **Closures capture the environment, not just the values.** This means if a closure modifies a captured variable, all closures sharing that environment see the change.

LoxFunction: A Callable Wrapper

A function in Lox is a **first-class value** that can be stored in variables, passed as arguments, returned from other functions, and called. To represent this, we need a runtime value that wraps the static function definition with its dynamic execution context.

Component	Description
`LoxFunction`	A runtime object representing a callable Lox function or closure

Data Structure Details:

Field Name	Type	Description
`declaration`	`Stmt.Function`	The AST node representing the function definition (contains name, parameters, body)
`closure`	`Environment`	A reference to the **defining environment**—the environment active when the function was declared
`isInitializer`	`boolean`	Flag indicating whether this function is a class's `init` method (affects `this` and return value behavior)

Interface Methods:

Method Name	Parameters	Returns	Description
`call`	`Interpreter interpreter, List<Object> arguments`	`Object`	Executes the function body with the given arguments, returning the function's result
`arity`	(none)	`int`	Returns the number of parameters the function expects
`toString`	(none)	`String`	Returns a string representation like "<fn name>"

The `LoxFunction` object serves as a **bridge** between the static function definition (in the AST) and the dynamic execution context (the environment chain). When a function is declared, we create a `LoxFunction` object, capturing the *current* environment as its `closure` field. This captured environment becomes the **parent** of the new environment created each time the function is called.

> **Key Design Insight:** The function's `closure` field is set once, when the function is **defined**, not when it's called. This is what enables lexical scoping—the function can access variables that existed when it was created, even if they're no longer in scope at the call site.

Function Call and Return Algorithm

Function execution involves several coordinated steps across the interpreter, the `LoxFunction` object, and the environment chain. Here's the complete algorithm:

1. Function Declaration Evaluation

When the interpreter encounters a function declaration statement (`Stmt.Function`):

1. **Create closure:** Capture a reference to the interpreter's *current* environment (the one active when the function is defined).
2. **Wrap in LoxFunction:** Instantiate a `LoxFunction` object with:
   - The function's AST node (`declaration`)
   - The captured environment (`closure`)
   - `isInitializer` = `false` (for regular functions; class methods set this differently)
3. **Bind to name:** Define the function's name in the current environment, mapping it to the `LoxFunction` object.

2. Function Call Evaluation

When evaluating a call expression (`Expr.Call`):

1. **Evaluate callee:** Evaluate the expression before the parentheses (could be a variable name, property access, or another call).
2. **Validate callee:** Ensure the callee is actually callable (an instance of `LoxFunction` , `LoxClass` , or a native function). If not, throw a runtime error.
3. **Evaluate arguments:** Evaluate each argument expression from left to right, collecting the resulting values.
4. **Check arity:** Verify the number of arguments matches the function's expected parameter count (`arity`). If not, throw a runtime error.
5. **Create call environment:** Instantiate a new `Environment` object with the function's `closure` as its parent.
6. **Bind parameters:** For each parameter name and corresponding argument value, define a binding in the new call environment.
7. **Execute body:**
   - Temporarily set the interpreter's current environment to the new call environment
   - Execute each statement in the function's body using `executeBlock`
   - Catch any `Return` exception thrown by a return statement
   - Restore the interpreter's previous environment
8. **Handle return:**
   - If a `Return` exception was caught, extract its value and return that as the call expression's result
   - If no return statement was encountered (or the function completes without hitting one), return `nil`

3. Return Statement Execution

When executing a return statement (`Stmt.Return`):

1. **Evaluate value:** If the return statement has a value expression, evaluate it. Otherwise, use `nil` .
2. **Throw Return exception:** Create and throw a custom `Return` exception containing the value. This exception propagates up through the call stack until caught by the function call logic.

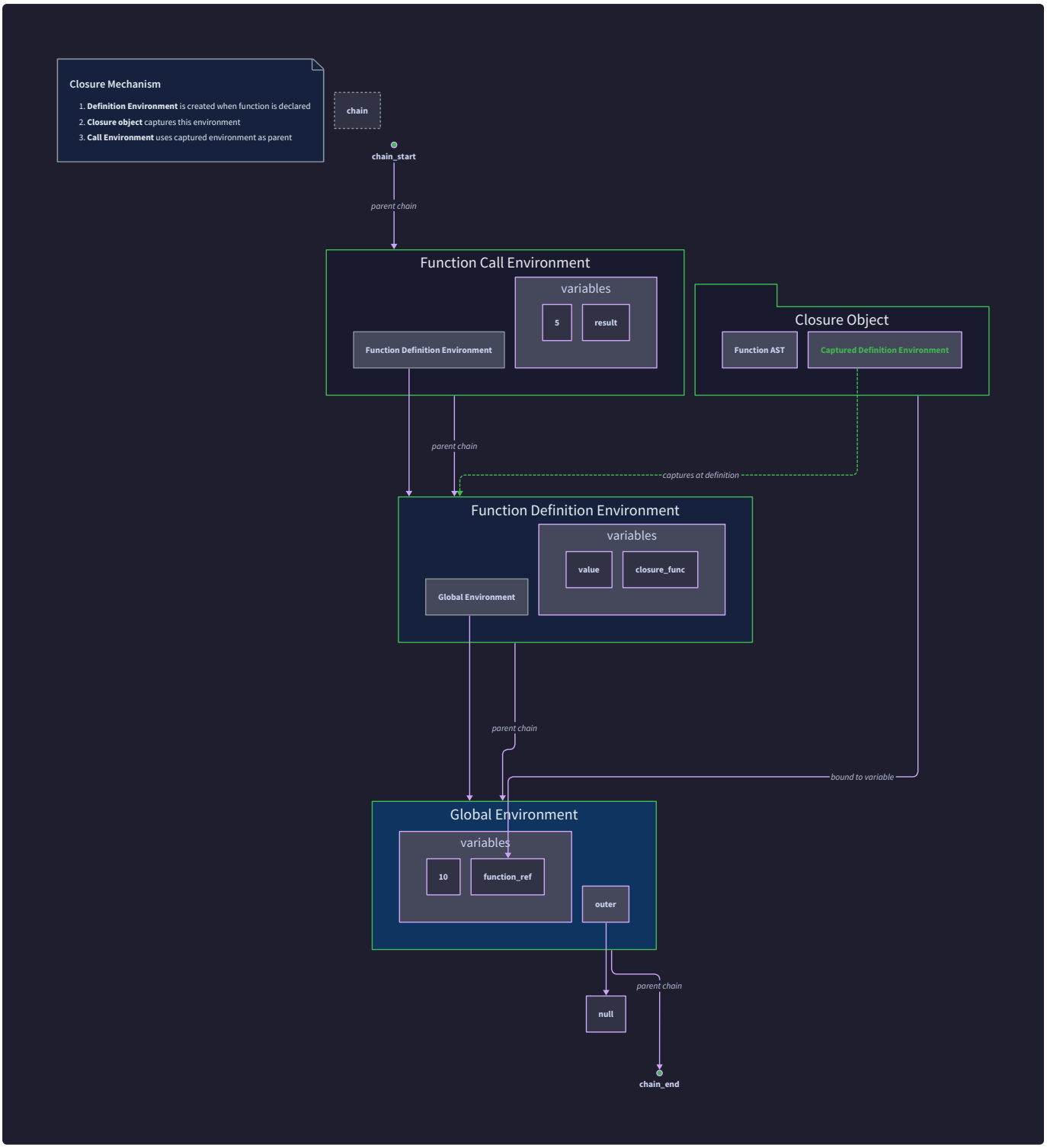
> The `Return` exception is a **control flow mechanism**, not an error. It provides a clean way to immediately exit nested statement execution and propagate a value back to the caller without unwinding the call stack manually.

ADR: Environment Capture for Closures

Decision: Capture Defining Environment by Reference

- **Context:** Functions in Lox use lexical scoping: an inner function can access variables from any outer scope where it's defined. We need to decide how to make these outer variables available when the function is called, possibly much later and from a different scope.
- **Options Considered:**
  1. **Capture defining environment by reference:** Store a pointer to the actual environment object active at function definition.
  2. **Capture defining environment by copy:** Create a snapshot copy of all variable bindings at definition time.
  3. **Use caller's environment:** Ignore lexical scoping and use the environment at the call site (dynamic scoping).
- **Decision:** We chose option 1—capture the defining environment by reference.
- **Rationale:**
  - **Correct semantics:** Capturing by reference allows closures to see *updates* to captured variables, which matches the behavior of most modern languages and is essential for implementing mutable state in functional patterns.
  - **Memory efficiency:** Copying the entire environment (which could be large) for every function definition would be wasteful, especially for deeply nested functions.
  - **Implementation simplicity:** A reference is trivial to store and creates the natural parent-child environment relationship needed for variable lookup.
  - **Dynamic scoping rejection:** Option 3 would violate Lox's specified lexical scoping rules and make programs much harder to reason about.
- **Consequences:**
  - Closures can modify captured variables, affecting other closures sharing the same environment.
  - The captured environment must stay alive as long as any closure references it (no garbage collection in our simple interpreter).
  - Variable lookup follows the natural parent chain: call environment → defining environment → its parent → ... → global.

Comparison of Environment Capture Strategies:


Closure Mechanism
1. Definition Environment is created when function is declared
2. Closure object captures this environment
3. Call Environment uses captured environment as parent
chain
chain_start
parent chain
Function Call Environment
variables
5
result
Function Definition Environment
Closure Object
Function AST
Captured Definition Environment
parent chain
captures at definition
Function Definition Environment
variables
value
closure_func
Global Environment
parent chain
bound to variable
Global Environment
variables
10
function_ref
outer
parent chain
null
chain_end

Option	Pros	Cons	Suitable For
Capture by reference	- Allows mutable captured variables - Memory efficient - Simple implementation	- Can create unintentional sharing - Requires environment lifetime management	Lexical scoping with mutable state (Lox, JavaScript, Python)
Capture by copy	- Isolates closure from changes - Easier reasoning about behavior	- Inefficient for large environments - Doesn't allow updates to outer variables	Immutable functional languages
Use caller's environment	- Extremely simple to implement - No capture needed	- Violates lexical scoping - Unpredictable variable resolution	Dynamic scoping languages (some Lisps, early Perl)

Common Pitfalls

⚠️ Pitfall: Not creating a new environment per function call

- **Description:** Reusing the same environment object for multiple calls to the same function.
- **Why it's wrong:** Breaks recursion and reentrancy. Each call needs its own parameter bindings; otherwise, a recursive call would overwrite the previous call's parameters, and returns would leave the environment in an inconsistent state.
- **Fix:** Always create a fresh `Environment` object inside `LoxFunction.call()`, with the closure as its parent.

⚠️ Pitfall: Incorrectly binding 'this' in methods

- **Description:** Forgetting to bind the `this` keyword to the instance when a method is called.
- **Why it's wrong:** Methods need access to their instance's fields via `this`. Without proper binding, `this` either resolves to nothing or resolves incorrectly.
- **Fix:** When calling a method, create an environment for the call with an extra binding: `"this" → instance`. This happens in the method call logic, not in regular function calls.

⚠️ Pitfall: Forgetting to propagate return values through the call stack

- **Description:** Catching the `Return` exception but not properly extracting and returning its value to the caller.
- **Why it's wrong:** Function calls would always return `nil` regardless of what the return statement specified.
- **Fix:** Ensure `LoxFunction.call()` catches the `Return` exception, extracts its `value` field, and returns that value as the call result.

⚠️ Pitfall: Memory leaks from captured environment chains

- **Description:** Environments captured by closures are never released, causing unbounded memory growth in long-running programs.
- **Why it's wrong:** In a production interpreter, this would eventually exhaust available memory. Each closure keeps its entire defining environment chain alive.
- **Fix:** In our educational interpreter, we accept this limitation. A production implementation would need a garbage collector to detect and reclaim unreachable environments.

⚠️ Pitfall: Confusing Java's return with Lox's return

- **Description:** Using Java's `return` statement inside the interpreter's visitor methods to implement Lox's return statement.
- **Why it's wrong:** A Java `return` only exits the current visitor method, not the entire Lox function body. The interpreter needs to unwind through potentially multiple nested statements.
- **Fix:** Use the custom `Return` exception thrown by `visitReturnStmt` and caught in `LoxFunction.call()`.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Function Representation	Single `LoxFunction` class with all fields	Separate interfaces for callables: `LoxCallable` with `call()` and `arity()`
Return Mechanism	Custom `Return` exception class	Special `ReturnValue` object with stack unwinding logic
Environment Capture	Direct reference to `Environment` object	Environment identifier with indirect lookup table

Recommended File/Module Structure:

```
lox/
  interpreter/
    Interpreter.java           # Updated with function visit methods
    Environment.java           # Already exists from Milestone 5
    LoxFunction.java           # NEW: Function/closure runtime representation
    Return.java                # NEW: Exception for return statements
  parser/
    # Existing files unchanged
  ast/
    # Existing files unchanged
  lox/
    Lox.java                   # Main entry point
```

Infrastructure Starter Code (Complete):

```java
// File: interpreter/Return.java

package com.craftinginterpreters.lox.interpreter;


/**

 * Internal exception used to implement return statements.

 * This is not an error but a control flow mechanism.

 */

public class Return extends RuntimeException {

  public final Object value;


  public Return(Object value) {

    super(null, null, false, false);  // Disable stack trace for performance

    this.value = value;

  }

}
```

Core Logic Skeleton Code:

JAVA

```java
// File: interpreter/LoxFunction.java

package com.craftinginterpreters.lox.interpreter;


import com.craftinginterpreters.lox.ast.*;

import java.util.List;


public class LoxFunction implements LoxCallable {

  private final Stmt.Function declaration;

  private final Environment closure;

  private final boolean isInitializer;


  public LoxFunction(Stmt.Function declaration, Environment closure,

                     boolean isInitializer) {

    this.declaration = declaration;

    this.closure = closure;

    this.isInitializer = isInitializer;

  }


  @Override

  public int arity() {

    // TODO 1: Return the number of parameters in the declaration

    return 0;

  }


  @Override

  public Object call(Interpreter interpreter, List<Object> arguments) {

    // TODO 2: Create a new environment for this function call

    //         The parent should be the captured closure, not the current environment



    // TODO 3: Bind each parameter name to its corresponding argument value

    //         Use environment.define(name.lexeme, value)



    try {

      // TODO 4: Execute the function body in the new environment

      //         Use interpreter.executeBlock(declaration.body, callEnvironment)

    } catch (Return returnValue) {

      // TODO 5: Handle return statement

      //         If this is an initializer, always return 'this' from closure
```

```
      //         Otherwise, return the value from the Return exception
      return null;
    }

    // TODO 6: Handle normal completion (no return statement reached)
    //         If this is an initializer, return 'this' from closure
    //         Otherwise, return nil
    return null;
  }

  @Override
  public String toString() {
    return "<fn " + declaration.name.lexeme + ">";
  }

  // TODO 7: Add a bind method for binding 'this' to instances (for methods)
  //         This will be used in the class implementation
}
```

JAVA

```java
// File: interpreter/Interpreter.java (additions only)

public class Interpreter implements Expr.Visitor<Object>, Stmt.Visitor<Void> {

  // ... existing fields and methods ...


  @Override

  public Object visitCallExpr(Expr.Call expr) {

    // TODO 1: Evaluate the callee expression (the part before parentheses)


    // TODO 2: Evaluate each argument expression, collecting values in a list


    // TODO 3: Verify the callee is callable (instanceof LoxCallable)

    //         If not, throw RuntimeError with expr.paren token


    // TODO 4: Check arity: arguments.size() must equal callee.arity()

    //         If not, throw RuntimeError with expr.paren token


    // TODO 5: Call the function: callee.call(this, arguments)

    //         Return the result

    return null;

  }


  @Override

  public Void visitFunctionStmt(Stmt.Function stmt) {

    // TODO 1: Create a LoxFunction with:

    //         - declaration = stmt

    //         - closure = current environment

    //         - isInitializer = false (for now)


    // TODO 2: Define the function name in the current environment

    //         environment.define(stmt.name.lexeme, function)

    return null;

  }


  @Override

  public Void visitReturnStmt(Stmt.Return stmt) {
```

```
    Object value = null;


    // TODO 1: If stmt.value is not null, evaluate it


    // TODO 2: Throw a Return exception with the value
    //         This will be caught in LoxFunction.call()
    throw new Return(value);
  }


  // Helper method for executing blocks (already exists, but ensure it supports functions)
  public void executeBlock(List<Stmt> statements, Environment environment) {
    Environment previous = this.environment;
    try {
      this.environment = environment;
      for (Stmt statement : statements) {
        execute(statement);
      }
    } finally {
      this.environment = previous;
    }
  }
}
```

Language-Specific Hints (Java):

- Use `List<Object>` for argument lists; Java's type erasure means you'll need to cast returned values.
- The `Return` exception should extend `RuntimeException` but disable stack trace generation for better performance since it's used for normal control flow.
- Consider creating a `LoxCallable` interface with `call()` and `arity()` methods to unify functions, classes, and future native functions.
- When implementing `executeBlock`, use a try-finally block to ensure the environment is always restored, even if an exception is thrown.

Milestone Checkpoint: After implementing functions, verify with this test:

```lox
// test.lox
fun makeCounter() {
  var i = 0;
  fun count() {
    i = i + 1;
    return i;
  }
  return count;
}

var counter = makeCounter();
print counter();  // Should print: 1
print counter();  // Should print: 2
```

Expected output:

```
1
2
```

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Function always returns nil	Return exception not caught or value not extracted	Add debug prints in `LoxFunction.call()` to see if Return is caught	Ensure `call()` catches `Return` and returns `returnValue.value`
Recursive function overwrites variables	Same environment reused for recursive calls	Check if you're creating new Environment in each `call()`	Create fresh `Environment` for each call with closure as parent
Closure can't see outer variables	Wrong environment captured as closure	Print closure's parent chain during function definition	Capture current environment when creating `LoxFunction`, not global
"Expected X arguments but got Y" error on correct call	Arity calculation wrong	Print `declaration.params.size()` in `arity()`	Return the correct parameter count
Function call evaluates arguments in wrong order	List evaluation order not left-to-right	Check order in `visitCallExpr` argument evaluation	Evaluate arguments sequentially before calling function

---

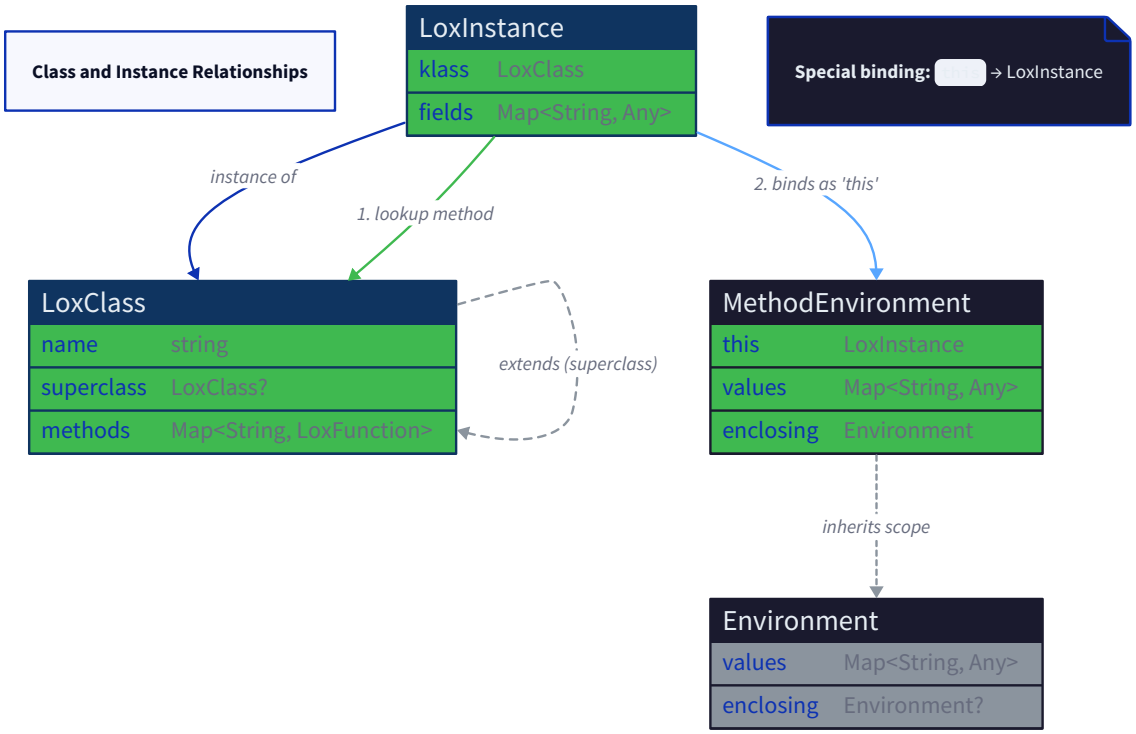
5.5 Component Design: Classes and Inheritance [Milestones 9 & 10]

> **Milestone(s):** Milestone 9 - Classes, Milestone 10 - Inheritance

This section extends the interpreter from a procedural language with functions to an object-oriented language with classes, instances, and inheritance. We'll design the runtime structures that represent classes and objects, implement method calls with implicit `this` binding, and add single inheritance with `super` calls. The key architectural challenge is modeling the relationship between classes, instances, and their environments while maintaining lexical scoping for closures and proper method resolution order.

Mental Model: The Class as a Blueprint and Factory

Think of a **class** as both a **blueprint** and a **factory**. The blueprint defines the structure—what methods the object will have—while the factory is a callable entity that produces new instances. An **instance** is a house built from the blueprint: it has its own storage rooms (fields) that can hold unique furnishings (values), but shares the architectural plans (methods) with every other house from the same blueprint. **Inheritance** is creating a modified copy of the blueprint: you start with the original blueprint, trace over it, and add or change some rooms while keeping the rest. When you build a house from this derived blueprint, it includes features from both the original and the modifications.



The `this` keyword is like a labeled floor plan inside the house that says "you are here"—it allows methods running within an instance to refer to the instance itself. The `super` keyword is a special phone line installed in the house that connects directly to the original blueprint's architect, allowing you to ask how a room was originally designed even if you've modified it in your copy.

LoxClass and LoxInstance Structures

We need two new runtime value types to represent classes and instances. These extend the existing `Object` type hierarchy (which already includes `LoxFunction`, `LoxString`, `LoxNumber`, etc.).

Type	Fields	Description
`LoxClass`	`name` (String)	The class's name as declared in source code
	`methods` (Map<String, LoxFunction>)	A map from method names to their corresponding `LoxFunction` objects
	`superclass` (LoxClass or `null`)	Reference to the superclass this class inherits from, if any
`LoxInstance`	`klass` (LoxClass)	Reference to the class this instance belongs to
	`fields` (Map<String, Object>)	A mutable map storing the instance's field names and their current values

Both `LoxClass` and `LoxInstance` must implement a `call()` method to be callable (classes are called as constructors, instances are not callable unless they have a `call()` method, which Lox doesn't support). `LoxClass`'s `call()` creates a new instance and invokes the initializer. `LoxInstance`'s `call()` should throw a runtime error.

Additionally, we need to extend the `LoxFunction` class from Milestone 8 with an `isInitializer` boolean flag to mark the special `init` method, which must return `this` instead of its explicit return value.

> **Design Insight:** Classes in Lox are first-class values—they can be stored in variables, passed as arguments, and returned from functions. This is why `LoxClass` is a runtime value type just like numbers and strings. The class declaration statement evaluates to a `LoxClass` object and stores it in the current environment.

Instantiation, Property Access, and Method Resolution

The lifecycle of a class involves declaration, instantiation, property access, and method invocation. The following numbered procedures describe each operation.

1. Class Declaration Evaluation

When the interpreter encounters a `Class` statement:

1. **Evaluate superclass (if present):** If the class has a superclass clause (`class Derived < Base`), evaluate the superclass expression (which must be a variable referencing a class). This yields a `LoxClass` object or throws an error if not a class.
2. **Create method map:** For each method in the class body, create a `LoxFunction` object. The function's **closure** is set to the *current environment* (where the class is being defined), capturing the surrounding lexical scope. Mark the function as an initializer if its name is `"init"`.
3. **Instantiate class object:** Create a new `LoxClass` with the class name, method map, and superclass reference.
4. **Bind to name:** Store the `LoxClass` object in the current environment under the class's name.

2. Instance Creation (Calling a Class)

When a class is called as a function (e.g., `MyClass()`):

1. **Create instance:** Instantiate a new `LoxInstance` with its `klass` field pointing to the called `LoxClass` and an empty `fields` map.
2. **Bind `this` :** Create a new environment whose *enclosing* environment is the **instance's class's method closure environment** (the environment captured when the class was defined). In this new environment, define a special variable named `"this"` bound to the instance.
3. **Invoke initializer:** Look up the `"init"` method in the class's `methods` map. If found:
   - Create a new `LoxFunction` bound to the instance by setting its closure to the environment created in step 2 (which has `this` bound).
   - Call this function with the provided arguments.
   - If the initializer returns a value, discard it (unless it's a special early return). Instead, automatically return `this` .
4. **Return instance:** The result of the class call is the new `LoxInstance` .

3. Property Access (Get Expression)

When evaluating `instance.property` :

1. **Evaluate object:** Evaluate the left-hand expression to obtain an `LoxInstance` . If it's not an instance, throw a runtime error.
2. **Check fields:** Look up the property name in the instance's `fields` map. If found, return the value.
3. **Check methods:** If not in fields, look up the method name in the instance's class's `methods` map (and superclass chain). If found, return that method **bound to the instance** (create a new `LoxFunction` with the same declaration but whose closure is an environment that has `this` bound to the current instance).
4. **Error:** If neither field nor method exists, throw a runtime error.

4. Property Assignment (Set Expression)

When evaluating `instance.property = value` :

1. **Evaluate object:** Evaluate the left-hand object expression to an `LoxInstance` .
2. **Evaluate value:** Evaluate the right-hand expression.
3. **Store in fields:** Insert or update the property name in the instance's `fields` map with the value.
4. **Return value:** The assignment expression returns the assigned value (consistent with Lox's assignment semantics).

5. Method Resolution with Inheritance

When a method is called on an instance:

1. **Find method:** Starting at the instance's class, look for the method name in its `methods` map. If not found, recursively search the superclass chain.
2. **Bind `this`:** When found, create a new environment whose parent is the method's original closure (captured at class definition time) and define `"this"` as the instance in this new environment. Return a new `LoxFunction` with this environment as its closure.
3. **Super calls:** For `super.method()`, the lookup starts in the *superclass* of the class where the surrounding method is defined (not the instance's class). This ensures calling the inherited version, not an override.

ADR: Storing a Dedicated 'super' Environment

> **Decision: Store Superclass Reference in a Dedicated Environment for Method Execution**
>
> - **Context:** When a method uses `super.method()`, we need to resolve `method` in the superclass's scope, not the current class's. The simplest approach is to pass both the current instance and the superclass reference through the call chain, but this complicates the function call interface. Another approach is to create a special environment that links to the superclass.
> - **Options Considered:**
>   1. **Pass superclass explicitly:** Add a `superclass` parameter to `LoxFunction.call()` and modify all call sites.
>   2. **Store superclass in a dedicated environment:** Create a special environment that sits between the method's closure and the instance-binding environment, containing a `"super"` variable pointing to the superclass.
>   3. **Compute superclass at runtime:** Walk the class hierarchy on every `super` call by starting from the instance's class and finding the surrounding method's class via lexical analysis.
> - **Decision:** Option 2—store a reference to the superclass in a dedicated environment that becomes part of the chain when methods are executed.
> - **Rationale:** This approach keeps the function call interface unchanged and leverages the existing environment chain mechanism. The `super` keyword behaves like a special variable that's accessible only within methods, similar to `this`. By storing it in an environment, we maintain lexical scoping consistency and avoid adding special cases to the function call protocol.
> - **Consequences:**
>   - ✅ `super` is naturally limited to method contexts (the environment only exists during method execution).
>   - ✅ The method resolution for `super` is efficient (direct reference).
>   - ✅ No changes to `LoxFunction.call()` signature.
>   - ❄️ Adds complexity to the environment chain construction for method calls.
>   - ❄️ Requires careful setup to ensure `super` points to the correct superclass when methods are inherited.

Option	Pros	Cons	Chosen?
Pass superclass explicitly	Simple implementation; Clear data flow	Pollutes call interface; Requires changes at all call sites	No
Store superclass in dedicated environment	**Leverages existing environment mechanism; Clean separation of concerns**	**Adds extra environment layer; More complex setup**	**Yes**
Compute superclass at runtime	No extra state needed; Dynamic resolution	Computationally expensive; Requires tracking lexical class context	No

The implementation creates a new environment (call it the "super environment") whose *enclosing* environment is the method's original closure. This super environment contains a single binding: `"super"` → `LoxClass` (the superclass). Then, the instance-binding environment (with `this`) has this super environment as its parent. This creates a three-layer chain: instance environment (has `this`) → super environment (has `super`) → method closure (captured lexical scope).

Common Pitfalls

⚠️ Pitfall: Forgetting to bind `this` in methods
Description: When a method is accessed (e.g., `instance.method`), returning the raw `LoxFunction` stored in the class without binding `this` to the specific instance.
Why it's wrong: The method's body references `this` expecting it to be the instance the method was called on, but without binding, `this`

resolves to whatever it was when the class was defined (likely `nil` or wrong object).
Fix: When a method is retrieved from a class (via property access), return a *new* `LoxFunction` that wraps the original function but with a closure that has `this` bound to the current instance.

⚠️ Pitfall: Improper initialization order (fields vs. init)
Description: Allowing field assignments in the `init` method to overwrite fields set before `init` runs, or vice versa.
Why it's wrong: The instance should be fully initialized after the constructor runs, with fields set either in `init` or directly. If the instance's `fields` map is not created before `init` runs, assignments inside `init` fail.
Fix: Create the instance's `fields` map *before* calling `init`. Inside `init`, assignments modify this existing map. This allows `init` to set default values or validate fields.

⚠️ Pitfall: Incorrect method resolution order (instance → class → superclass)
Description: When looking up a method, checking the instance's fields first, then the class's methods, but forgetting to walk the superclass chain for inherited methods.
Why it's wrong: Subclass instances cannot call inherited methods, breaking inheritance.
Fix: Implement recursive method lookup: start at the instance's class, search its method map; if not found, recursively search its superclass.

⚠️ Pitfall: Cycles in inheritance
Description: Allowing a class to inherit from itself directly (`class A < A`) or indirectly (`class A < B; class B < A`).
Why it's wrong: Infinite loops during method resolution or instance creation.
Fix: During class declaration, validate that the superclass is not the class itself and that no cycle exists in the inheritance chain (simple cycle detection by walking the superclass chain).

⚠️ Pitfall: Not handling `super` outside of a class context
Description: Allowing `super` keyword to be used outside of a method (e.g., in top-level code).
Why it's wrong: `super` only makes sense within a method to call a superclass method. Using it elsewhere is a semantic error.
Fix: During parsing, `super` is allowed anywhere (as it's an expression), but at runtime, throw a clear error if `super` is evaluated outside of a method context (i.e., when there's no `"super"` variable in the environment chain).

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Class representation	`LoxClass` and `LoxInstance` as plain Java classes with fields	Use interfaces `LoxCallable` and `LoxObject` for uniformity
Method binding	Create new `LoxFunction` with modified closure on each property get	Cache bound methods per instance to avoid recreation
Inheritance chain	Store `superclass` reference and walk recursively	Precompute method table per class (vtable) for faster dispatch

Recommended File/Module Structure:

```
src/
└── com/
    └── craftinginterpreters/
        └── lox/
            ├── AstPrinter.java          (existing)
            ├── Environment.java         (existing)
            ├── Interpreter.java         (extended)
            ├── Lox.java                 (existing)
            ├── Parser.java              (existing)
            ├── Scanner.java             (existing)
            ├── Token.java               (existing)
            ├── TokenType.java           (existing)
            ├── ast/                     (existing)
            │   ├── Expr.java            (extended with Get, Set, This, Super)
            │   └── Stmt.java            (extended with Class)
            └── runtime/
                ├── LoxCallable.java     (interface)
                ├── LoxFunction.java     (extended with isInitializer)
                ├── LoxClass.java        (new)
                └── LoxInstance.java     (new)
```

Infrastructure Starter Code:

First, extend the `LoxFunction` class to support initializers:

JAVA

```java
package com.craftinginterpreters.lox.runtime;

import com.craftinginterpreters.lox.Interpreter;
import com.craftinginterpreters.lox.ast.Stmt;
import com.craftinginterpreters.lox.Environment;
import java.util.List;

public class LoxFunction implements LoxCallable {
    private final Stmt.Function declaration;
    private final Environment closure;
    private final boolean isInitializer;

    public LoxFunction(Stmt.Function declaration, Environment closure,
                       boolean isInitializer) {
        this.declaration = declaration;
        this.closure = closure;
        this.isInitializer = isInitializer;
    }

    @Override
    public int arity() {
        return declaration.params.size();
    }

    @Override
    public Object call(Interpreter interpreter, List<Object> arguments) {
        Environment environment = new Environment(closure);
        for (int i = 0; i < declaration.params.size(); i++) {
            environment.define(declaration.params.get(i).lexeme, arguments.get(i));
        }

        try {
            interpreter.executeBlock(declaration.body, environment);
        } catch (Return returnValue) {
            // Special case: init() always returns 'this'
            if (isInitializer) return closure.getAt(0, "this");
            return returnValue.value;
        }
```

```
        // If no return statement, init() returns 'this', others return nil
        if (isInitializer) return closure.getAt(0, "this");
        return null;
    }

    @Override
    public String toString() {
        return "<fn " + declaration.name.lexeme + ">";
    }

    // Bind this function to a specific instance
    public LoxFunction bind(LoxInstance instance) {
        Environment environment = new Environment(closure);
        environment.define("this", instance);
        return new LoxFunction(declaration, environment, isInitializer);
    }
}
```

Core Logic Skeleton Code:

1. **LoxClass.java** (complete starter):

JAVA

```java
package com.craftinginterpreters.lox.runtime;

import com.craftinginterpreters.lox.Interpreter;
import java.util.List;
import java.util.Map;

public class LoxClass implements LoxCallable {
    public final String name;
    public final LoxClass superclass;
    private final Map<String, LoxFunction> methods;

    public LoxClass(String name, LoxClass superclass,
                    Map<String, LoxFunction> methods) {
        this.name = name;
        this.superclass = superclass;
        this.methods = methods;
    }

    @Override
    public String toString() {
        return name;
    }

    @Override
    public Object call(Interpreter interpreter, List<Object> arguments) {
        // TODO 1: Create a new instance of this class
        // TODO 2: Find the init method in the class's methods
        // TODO 3: If init exists, bind it to the instance and call it with arguments
        // TODO 4: Return the instance (init's return value is ignored, instance is returned)
        return null;
    }

    @Override
    public int arity() {
        // TODO: Return the arity of the init method, or 0 if no init
        return 0;
    }

    public LoxFunction findMethod(String name) {
```

```
        // TODO 1: Check if method exists in this class's methods map
        // TODO 2: If not found and superclass exists, recursively search superclass
        // TODO 3: Return the method or null if not found
        return null;
    }
}
```

2. **LoxInstance.java** (complete starter):

```java
package com.craftinginterpreters.lox.runtime;

import java.util.HashMap;
import java.util.Map;

public class LoxInstance {
    private final LoxClass klass;
    private final Map<String, Object> fields = new HashMap<>();

    public LoxInstance(LoxClass klass) {
        this.klass = klass;
    }

    @Override
    public String toString() {
        return klass.name + " instance";
    }

    public Object get(String name) {
        // TODO 1: Check if field exists in the fields map, return if found
        // TODO 2: If not a field, look up method in the class
        // TODO 3: If method found, return it bound to this instance
        // TODO 4: Throw a runtime error if neither field nor method exists
        return null;
    }

    public void set(String name, Object value) {
        // TODO: Store the value in the fields map
    }
}
```

3. **Interpreter.java extensions** (skeleton for new visitor methods):

JAVA

```java
// Add to Interpreter class:

@Override
public Void visitClassStmt(Stmt.Class stmt) {
    // TODO 1: Evaluate superclass expression (if present) and ensure it's a LoxClass
    // TODO 2: Create a map of method names to LoxFunctions
    //   - For each method, create a LoxFunction with closure = current environment
    //   - Mark as initializer if method name is "init"
    // TODO 3: Create a new LoxClass with name, superclass, and methods
    // TODO 4: Define the class name in the current environment
    return null;
}

@Override
public Object visitGetExpr(Expr.Get expr) {
    // TODO 1: Evaluate the object expression (must be a LoxInstance)
    // TODO 2: Call instance.get(propertyName) to get field or bound method
    // TODO 3: Return the result
    return null;
}

@Override
public Object visitSetExpr(Expr.Set expr) {
    // TODO 1: Evaluate the object expression (must be a LoxInstance)
    // TODO 2: Evaluate the value expression
    // TODO 3: Call instance.set(propertyName, value)
    // TODO 4: Return the value
    return null;
}

@Override
public Object visitThisExpr(Expr.This expr) {
    // TODO 1: Look up "this" in the environment chain
    // TODO 2: Return the value (must be a LoxInstance)
    return null;
}

@Override
```

```
public Object visitSuperExpr(Expr.Super expr) {

    // TODO 1: Retrieve the "super" variable from the environment (should be a LoxClass)

    // TODO 2: Retrieve the "this" variable from the environment (should be a LoxInstance)

    // TODO 3: Find the method in the superclass (not the current class)

    // TODO 4: Return the method bound to the current instance

    return null;

}
```

Language-Specific Hints (Java):

- Use `HashMap<String, Object>` for the fields map in `LoxInstance` and methods map in `LoxClass`.
- The `bind` method in `LoxFunction` creates a shallow copy of the function with a new closure—this is efficient and maintains the original function's code.
- When throwing runtime errors for invalid property access, use the token's line number from the AST node for error reporting.
- For the `super` environment, create a new `Environment` with a single binding: `environment.define("super", superclass)`. This environment's parent should be the method's original closure.

Milestone Checkpoint: After implementing classes (Milestone 9), test with this Lox program:

```lox
class Breakfast {
  init(meat, bread) {
    this.meat = meat;
    this.bread = bread;
  }

  serve(who) {
    print "Enjoy your " + this.meat + " and " + this.bread + ", " + who + ".";
  }
}

var baconAndToast = Breakfast("bacon", "toast");
baconAndToast.serve("Dear Reader");
```

Expected output: `Enjoy your bacon and toast, Dear Reader.`

After implementing inheritance (Milestone 10), test with:

```lox
class A {
  method() {
    print "A.method()";
  }
}

class B < A {
  method() {
    print "B.method()";
  }

  test() {
    super.method();
  }
}

var b = B();
b.test();  // Should print "A.method()"
```

Expected output: `A.method()`

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
`this` is `nil` inside a method	Method not bound to instance	Check if `visitGetExpr` calls `bind()` on the retrieved method	Ensure property get returns `method.bind(instance)`
`super` call says "superclass method not found"	Super environment not set up	Print environment chain during method call to see if `"super"` exists	Make sure the super environment is created and linked when a method is bound
Instance fields disappear after init	Fields map reinitialized	Check if `init` creates a new map instead of using instance's map	Ensure instance's fields map is created once and passed to init
Inherited method calls subclass override	Method lookup starts at wrong class	Trace `findMethod` calls; should start at superclass for `super`	In `visitSuperExpr`, look up method in superclass, not instance's class

---

6. Interactions and Data Flow

> **Milestone(s):** All milestones (Milestones 1-10) — This section demonstrates how the components defined in previous design sections collaborate to transform inert source code into executed behavior. Understanding these interactions is critical for debugging and for appreciating the architecture as a cohesive system.

This section traces the complete journey of a Lox program from a string of characters to executed side effects and printed output. We'll follow the data as it flows through the pipeline of components—**Scanner**, **Parser**, and **Interpreter**—and examine the concrete data structures that carry meaning at each stage. Think of this as tracing a package through a sophisticated logistics network: the **Scanner** scans the shipping label (source code) and produces a detailed packing list (tokens), the **Parser** inspects that list and builds a precise loading plan (AST), and the **Interpreter** executes that plan by moving goods (runtime values) between warehouses (environments) to deliver the final outcome.

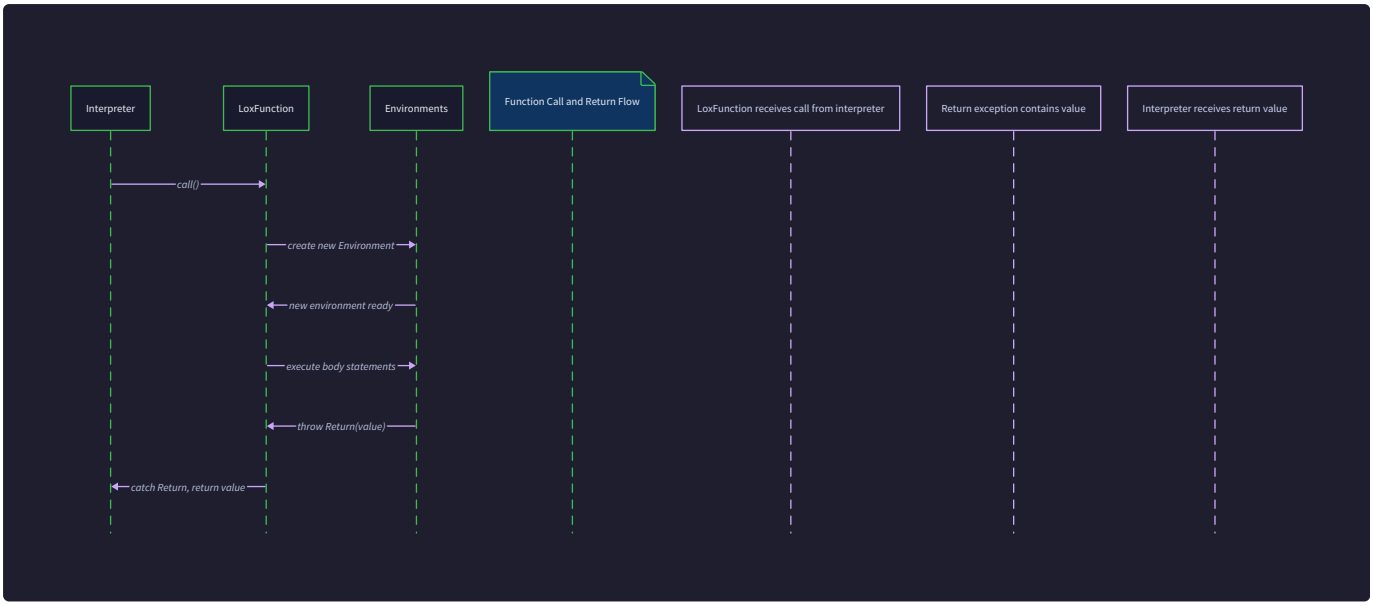
End-to-End Sequence for a Sample Program

Let's trace the complete execution of a simple Lox program: `var x = 1 + 2; print x;`. This program declares a variable, initializes it with the result of an arithmetic expression, and then prints the variable's value. The following numbered steps detail the transformation at each stage of the interpretation pipeline.

1. **Source Code Input**: The program begins as a plain Java `String`: `"var x = 1 + 2; print x;"`. This string is passed to the main entry point, typically `Lox.run()`.
2. **Lexical Analysis (Scanner)**: The `Scanner` receives the source string and begins its character-by-character scan. Its internal state (`start`, `current`, `line`) advances, identifying lexical boundaries.
    - It skips the initial whitespace, then recognizes `var` as a keyword, producing a `Token` with `type = VAR`, `lexeme = "var"`, `literal = null`, `line = 1`.
    - It advances, recognizes `x` as an identifier, producing a token with `type = IDENTIFIER`, `lexeme = "x"`, `literal = null`, `line = 1`.
    - It recognizes `=` as a single-character operator (`EQUAL`).
    - It recognizes `1` as a number literal, converting the substring `"1"` to a Java `Double` value `1.0` stored in the token's `literal` field (`type = NUMBER`, `lexeme = "1"`, `literal = 1.0`).
    - It recognizes `+` as an operator (`PLUS`).
    - It recognizes `2` as another number literal (`NUMBER` with `literal = 2.0`).
    - It recognizes `;` as a delimiter (`SEMICOLON`).
    - It then processes the second statement similarly: `print` becomes a `PRINT` keyword token, `x` becomes another `IDENTIFIER` token, and the final `;` becomes a `SEMICOLON`.
    - Finally, it appends an `EOF` token to signal the end of the token stream. The `Scanner` returns a `List<Token>` containing these 9 tokens (excluding whitespace).

3. **Syntactic Analysis (Parser)**: The `Parser` receives the `List<Token>` and begins its recursive descent parse, starting with the `program()` rule which calls `declaration()` repeatedly.
    - For the first statement, `declaration()` matches the `VAR` token and invokes `varDeclaration()`. This method consumes the `IDENTIFIER("x")` token, expects and consumes the `EQUAL` token, then calls `expression()` to parse the initializer.
    - Parsing the expression `1 + 2` involves the precedence cascade: `expression()` → `equality()` → `comparison()` → `term()` → `factor()` → `unary()` → `primary()`. At the `primary()` level, the number literal `1` is recognized, creating a `Literal` expression node with `value = 1.0`. The parser backtracks: at the `term()` level, it sees the `PLUS` token. Since `+` is handled at the `term` precedence level, it continues parsing the right-hand side, resulting in another `Literal` for `2`. It then constructs a `Binary` expression node with `left = Literal(1.0)`, `operator = Token(PLUS, "+")`, `right = Literal(2.0)`.
    - The `varDeclaration()` method thus creates a `Var` statement node with `name = Token(IDENTIFIER, "x")` and `initializer = Binary(Literal(1.0), PLUS, Literal(2.0))`.
    - The parser then synchronizes at the `SEMICOLON`.
    - For the second statement, `declaration()` sees `PRINT` and calls `printStatement()`. This method consumes the `PRINT` token, parses the following expression (which is just a `Variable` expression node referencing the token `IDENTIFIER("x")`), expects the `SEMICOLON`, and returns a `Print` statement node with `expression = Variable(Token(IDENTIFIER, "x"))`.
    - The `Parser` returns a `List<Stmt>` containing two elements: first the `Var` statement, then the `Print` statement.
4. **Semantic Analysis and Execution (Interpreter)**: The `Interpreter` receives the `List<Stmt>` via its `interpret()` method. It creates or reuses a global `Environment` (a chain of variable name-to-value maps). It then iterates through the statements, calling `execute()` on each.
    - **Executing `Var x = 1 + 2;`**: The `visitVarStmt()` method is invoked with the `Var` node. It first evaluates the initializer expression by calling `evaluate()` on the `Binary` node.
        - `evaluate()` on the `Binary` node recursively evaluates its left operand (`Literal(1.0)`) → returns `1.0`.
        - It evaluates the right operand (`Literal(2.0)`) → returns `2.0`.
        - It checks the operator (`PLUS`) and, since both operands are numbers, performs arithmetic addition, producing the runtime value `3.0` (a Java `Double`).
    - The interpreter then calls `environment.define("x", 3.0)`, which inserts the binding `"x" → 3.0` into the current environment's `values` map.
    - **Executing `print x;`**: The `visitPrintStmt()` method is invoked. It evaluates its expression—a `Variable` node.
        - `evaluate()` on the `Variable` node calls `environment.get(Token(IDENTIFIER, "x"))`. The environment looks up `"x"` in its map and returns the value `3.0`.
    - The interpreter passes this value to its `stringify()` helper, which converts the Double `3.0` to the string `"3"`. This string is then written to standard output (e.g., `System.out.println("3")`).
5. **Program Completion**: The interpreter finishes executing the statement list. The `interpret()` method returns void. The program's side effect—printing `"3"`—has been achieved. The runtime values (`3.0` bound to `x`) may persist in the global environment if the interpreter continues (e.g., in a REPL), but for this script execution, they are discarded as the process ends.

For a more complex program involving functions and closures, the data flow extends further. Consider a brief excerpt: `fun makeCounter() { var i=0; return fun() { i = i + 1; return i; }; }`.



illustrates the interactions during a function call. The **Parser** produces a `Function` statement node for `makeCounter`. When the **Interpreter** executes this declaration, it creates a `LoxFunction` object that captures the *current* environment as its `closure`. Later, when `makeCounter()` is called, `LoxFunction.call()` creates a *new* environment whose *enclosing* link is that captured closure. The inner function's `LoxFunction` is created within this new environment, capturing a reference to it. When the inner function is later called, its execution environment chains back through that captured link, allowing it to access and modify the `i` variable that outlives the execution of `makeCounter`. This environment chain is the mechanism that enables lexical scoping and closures.

Internal Data Handoffs

The transformation pipeline is defined by specific, immutable data structures passed between components. Each handoff point represents a complete shift in perspective on the program: from characters to words, from words to grammatical structure, from structure to executable meaning.

The following table details the primary data structures at each stage of the pipeline, their role, and the component responsible for producing them.

Pipeline Stage	Input Data Structure	Output Data Structure	Producing Component	Description of Transformation
Lexical Analysis	`String` (source code)	`List<Token>`	`Scanner`	Converts a linear sequence of characters into a linear sequence of categorized tokens. Whitespace and comments are filtered out. Each token carries its source location (`line`) and, for literals, a cooked value (`literal`).
Syntactic Analysis	`List<Token>`	`List<Stmt>` (a program AST)	`Parser`	Converts a flat sequence of tokens into a hierarchical tree of syntactic nodes (the AST). The tree structure implicitly encodes operator precedence, grouping, and the nesting of statements within blocks, functions, and control flow bodies.
Semantic Analysis & Execution	`List<Stmt>` (AST)	Side Effects & Program Output	`Interpreter` (with `Environment`)	Recursively traverses the AST, evaluating expressions to produce runtime values (`Object` s) and executing statements to modify the environment (variable bindings) and produce side effects (e.g., printing to console). The `Environment` chain is the primary mutable runtime state.

The flow of control and data during interpretation is more intricate than a simple linear pipeline. The following numbered procedure outlines the key handoffs and decision points within the Interpreter's core execution loop.

1. **Initialization**: The `Interpreter` is initialized with a `globals` environment. At the start of `interpret(List<Stmt>)` , it sets `environment = globals` .

2. **Statement Iteration**: For each `Stmt` in the list:
    - The interpreter calls `stmt.accept(this)`. This invokes the appropriate `visit*Stmt` method via the Visitor pattern's double dispatch.
    - Each `visit*Stmt` method may call `evaluate(Expr)` on embedded expression nodes, which in turn triggers `expr.accept(this)` and the corresponding `visit*Expr` methods.
    - This creates a recursive descent through the AST that mirrors the syntactic structure but is driven by execution semantics.
3. **Expression Evaluation Handoff**: When `evaluate(Expr)` is called on an expression node:
    - **Literal**: Returns the pre-computed `value` object (e.g., `Double`, `String`).
    - **Variable**: Handles off to `environment.get(nameToken)` to retrieve the current value bound to that name in the environment chain.
    - **Assign**: First `evaluate()`s the right-hand side value, then hands off to `environment.assign(nameToken, value)` to update the binding.
    - **Binary**: `evaluate()`s left and right operands, then performs type checking and the appropriate operation (e.g., arithmetic, comparison), returning a new runtime value.
4. **Environment Chain Traversal**: The `Environment.get()` and `Environment.assign()` methods implement the **handoff between scopes**. They first check the current environment's `values` map. If the identifier is not found, they recursively delegate to the `enclosing` environment (if not `null`). This chain of handoffs continues until the variable is found (success) or the global scope is reached without success (runtime error).
5. **Function Call Handoff**: A function call (`Expr.Call`) represents a major handoff:
    - The `callee` expression is evaluated to a `LoxFunction` (or `LoxClass`).
    - Argument expressions are evaluated to produce a `List<Object>` of runtime values.
    - Control hands off to `LoxFunction.call(interpreter, arguments)`. This method:
        - Creates a new `Environment` whose `enclosing` link is the function's captured `closure`.
        - Defines parameters in this new environment, mapping names to argument values.
        - Pushes this environment onto the interpreter's scope stack (`executeBlock()` with the new environment).
        - Executes the function's body statements. A `Return` statement throws a custom exception to unwind the call stack and hand the return value back to the caller.
6. **Output Handoff**: The `Print` statement's final handoff is to the host platform's standard output stream. The interpreter's `stringify()` method converts a Lox runtime `Object` to a Java `String`, which is then passed to `System.out.println()`.

> **Key Insight:** The data handoffs are not merely passing data forward; they also establish **responsibility boundaries**. The Scanner is responsible for *lexical correctness*, the Parser for *syntactic correctness*, and the Interpreter for *semantic (runtime) correctness*. Errors detected at each stage are reported using the most appropriate data available: the Scanner reports character and line numbers, the Parser reports unexpected tokens, and the Interpreter reports runtime values and operation types.

Implementation Guidance

This section bridges the architectural description to concrete Java code. It provides a skeleton for the main driver class that orchestrates the component interactions and shows how the data flows through method calls.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Main Driver	Single `Lox` class with static methods	Separate `Lox` (frontend), `Interpreter` (backend), and `Runner` (orchestrator) classes
Error Reporting	Print errors to `System.err` with line numbers	Collect errors in a list, allow for multiple error reporting, support IDE integration
Output Stream	Direct `System.out` and `System.err`	Configurable `PrintStream` for output and error for easier testing

B. Recommended File/Module Structure

Place the main orchestration logic in the root package, with components in their respective packages.

```
lox/
├── Lox.java                    # Main driver class
├── scanner/
│   ├── Scanner.java
│   └── Token.java
├── parser/
│   ├── Parser.java
│   └── ParseError.java
├── ast/
│   ├── Expr.java
│   ├── Stmt.java
│   └── (Visitor interfaces)
└── interpreter/
    ├── Interpreter.java
    ├── Environment.java
    ├── RuntimeError.java
    ├── LoxFunction.java
    ├── LoxClass.java
    └── LoxInstance.java
```

C. Infrastructure Starter Code

The `Lox` class serves as the entry point and orchestrator. Here is a complete, working version that handles file and REPL input, coordinates the components, and reports errors.

JAVA

```java
package com.craftinginterpreters.lox;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

public class Lox {
    private static final Interpreter interpreter = new Interpreter();
    static boolean hadError = false;
    static boolean hadRuntimeError = false;

    public static void main(String[] args) throws IOException {
        if (args.length > 1) {
            System.out.println("Usage: jlox [script]");
            System.exit(64);
        } else if (args.length == 1) {
            runFile(args[0]);
        } else {
            runPrompt();
        }
    }

    private static void runFile(String path) throws IOException {
        byte[] bytes = Files.readAllBytes(Paths.get(path));
        run(new String(bytes, Charset.defaultCharset()));
        if (hadError) System.exit(65);
        if (hadRuntimeError) System.exit(70);
    }

    private static void runPrompt() throws IOException {
        InputStreamReader input = new InputStreamReader(System.in);
        BufferedReader reader = new BufferedReader(input);

        for (;;) {
```

```
            System.out.print("> ");
            String line = reader.readLine();
            if (line == null) break;
            run(line);
            hadError = false; // Reset error flag for REPL
        }
    }

    private static void run(String source) {
        // 1. LEXICAL ANALYSIS: Source String -> Tokens
        Scanner scanner = new Scanner(source);
        List<Token> tokens = scanner.scanTokens();

        // 2. SYNTACTIC ANALYSIS: Tokens -> AST
        Parser parser = new Parser(tokens);
        List<Stmt> statements = parser.parse();

        // Stop if there was a parse error.
        if (hadError) return;

        // 3. SEMANTIC ANALYSIS & EXECUTION: AST -> Side Effects
        interpreter.interpret(statements);
    }

    // Error reporting methods (used by Scanner, Parser, Interpreter)
    static void error(int line, String message) {
        report(line, "", message);
    }

    private static void report(int line, String where, String message) {
        System.err.println("[line " + line + "] Error" + where + ": " + message);
        hadError = true;
    }

    static void error(Token token, String message) {
        if (token.type == TokenType.EOF) {
            report(token.line, " at end", message);
        } else {
            report(token.line, " at '" + token.lexeme + "'", message);
```

```
        }
    }

    static void runtimeError(RuntimeError error) {
        System.err.println(error.getMessage() + "\n[line " + error.token.line + "]");
        hadRuntimeError = true;
    }
}
```

D. Core Logic Skeleton Code

The `Interpreter.interpret()` method is the heart of the execution phase. Below is its skeleton with TODOs that map to the steps described in the data flow.

```java
public class Interpreter implements Expr.Visitor<Object>, Stmt.Visitor<Void> {

    final Environment globals = new Environment();

    private Environment environment = globals;


    // Main entry point for execution

    public void interpret(List<Stmt> statements) {

        try {

            // TODO 1: Iterate through each statement in the provided list

            // TODO 2: For each statement, call execute(stmt)

        } catch (RuntimeError error) {

            Lox.runtimeError(error);

        }

    }


    // Execute a single statement

    private void execute(Stmt stmt) {

        // TODO: Use the Visitor pattern: stmt.accept(this)

    }


    // Evaluate an expression to a runtime value

    Object evaluate(Expr expr) {

        // TODO: Use the Visitor pattern: expr.accept(this)

        return null;

    }


    // ... visitor method implementations will go here ...

}
```

E. Language-Specific Hints

- **Error Handling**: Use custom exception classes (`ParseError` , `RuntimeError`) to separate error reporting from control flow. `ParseError` can be caught internally by the Parser for synchronization, while `RuntimeError` is caught at the top level in `interpret()` .
- **Visitor Pattern Boilerplate**: Consider using an IDE or script to generate the visitor interface methods for the many `Expr` and `Stmt` subclasses.
- **REPL and File Execution**: The `runPrompt()` method uses `readLine()` which provides line editing history on many systems. For file execution, read the entire file into a string for simplicity.

F. Milestone Checkpoint

After implementing the core pipeline (through Milestone 5), you can verify the end-to-end flow with the sample program:

1. **Create a test file** `test.lox` with content: `var x = 1 + 2; print x;`
2. **Run your interpreter**: `java com.craftinginterpreters.lox.Lox test.lox`
3. **Expected Output**: The number `3` printed on a line by itself.

4. **Signs of Trouble**:
    - No output or an error: Check that your `Scanner` is producing the correct token list (use a debug print). Verify the `Parser` builds the expected AST (use an AST printer). Ensure the `Interpreter`'s `visitVarStmt` and `visitPrintStmt` are being called.
    - Incorrect output (e.g., `1`, `2`, `12`): Likely an issue in expression evaluation precedence or in the `Binary` expression evaluation logic (check operator handling).

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Program prints nothing	Parser failed silently due to error, `hadError` is true, `interpret()` not called.	Add a debug print in `run()` after parsing to see if `statements` is null/empty. Check `hadError` flag.	Ensure error reporting sets `hadError` and that `run()` returns early if `hadError` is true.
"Undefined variable 'x'" at runtime	Variable binding not stored or retrieved correctly from the environment.	Print the environment's `values` map after `define()` and before `get()`. Check that the environment chain is correct.	Verify `environment.define()` is called with the correct name string and value. Ensure `environment.get()` is searching the chain via the `enclosing` field.
Incorrect arithmetic result (e.g., 1+2=12)	Operands treated as strings, concatenated instead of added.	Print the types (`instanceof`) of the left and right operands in `visitBinaryExpr`.	Implement runtime type checking: ensure both operands are `Double` before performing arithmetic.
Parser goes into infinite loop	Grammar rule left-recursion or missing advance over tokens.	Add a debug print at the start of each parsing method showing the current token.	Ensure each parsing method that consumes a token calls `advance()` or `consume()`. Check for left-recursive grammar rules (not a problem in our recursive descent formulation if written correctly).

7. Error Handling and Edge Cases

> **Milestone(s):** All milestones (Milestones 1-10) — Error handling is a cross-cutting concern that evolves through each phase of interpretation, from lexical analysis through runtime execution.

Robust error handling is what separates a toy interpreter from a usable programming tool. In a tree-walking interpreter, errors can occur at three distinct phases: **lexical** (scanning), **syntactic** (parsing), and **semantic** (runtime). Each phase requires different detection, reporting, and recovery strategies. This section defines a comprehensive approach that provides clear, actionable feedback to Lox programmers while maintaining interpreter stability.

Error Classification and Reporting

Mental Model: The Error Lifecycle Think of error handling as a quality control pipeline in a manufacturing plant. At the first station (Scanner), raw materials (characters) are inspected for obvious defects like unrecognized symbols or malformed parts. Defective items are rejected immediately with a detailed defect report. At the second station (Parser), the arrangement of approved parts is checked against assembly diagrams (grammar rules); misassemblies trigger a pause while the assembly line resets. Finally, at the third station (Interpreter), the assembled product is tested under real-world conditions; operational failures (like trying to use a hammer as a screwdriver) halt production entirely with a clear explanation of what went wrong.

Error Categories and Detection Points

Error Category	Detection Point	Trigger Condition	Example
Lexical Error	`Scanner.scanTokens()`	Unrecognized character, unterminated string, malformed number	`"Hello, world!` (missing closing quote)
Syntactic Error	`Parser.parse()`	Token sequence violates grammar rules	`var x = ;` (missing expression after `=`)
Runtime Error	`Interpreter.evaluate()` / `execute()`	Semantic violation during execution	`"text" - 1` (invalid operands for `-`)

Error Reporting Standards

All errors must include:

1. **Location**: The exact line number (and ideally column) in the source file
2. **Phase**: Clear indication of whether it's a scanning, parsing, or runtime error
3. **Specific Message**: A human-readable description of what went wrong
4. **Context**: The offending token or expression that triggered the error

> **Key Insight:** Good error messages don't just say *what* went wrong; they help the programmer understand *why* and *how to fix it*. A message like "Unexpected ';' at line 5, expected expression" is far more actionable than simply "Syntax error."

Error Class Hierarchy

The interpreter uses a three-tiered exception hierarchy:

Exception Class	Superclass	When Thrown	Recovery Strategy
`ScanError`	`RuntimeException`	During lexical analysis when encountering invalid characters or unterminated strings	Report and continue scanning (skip invalid token)
`ParseError`	`RuntimeException`	During parsing when token sequence violates grammar	Enter panic mode, synchronize, continue parsing
`RuntimeError`	`RuntimeException`	During evaluation when semantic rules are violated	Propagate up, halt current statement execution

ADR: Unified vs. Phase-Specific Error Types

Decision: Phase-Specific Error Types

- **Context**: We need to handle errors differently based on which phase of interpretation encounters them. The scanner can often recover and continue, the parser uses panic-mode recovery, while runtime errors typically halt execution of the current statement.
- **Options Considered**:
  1. **Single `LoxError` class**: One exception type for all phases, with an enum field for phase.
  2. **Phase-specific subclasses**: Separate classes for `ScanError`, `ParseError`, and `RuntimeError`.
- **Decision**: Use phase-specific exception subclasses.
- **Rationale**:
  - **Type Safety**: The compiler ensures each phase only throws appropriate errors.
  - **Clear Intent**: Code that catches `ParseError` explicitly states it's handling parsing failures.
  - **Recovery Separation**: Each phase can implement recovery logic specific to its error type without complex conditional logic.
- **Consequences**:
  - Slightly more class definitions.
  - Clearer code organization and error handling flow.
  - Enables phase-specific recovery strategies.

Option	Pros	Cons	Chosen?
Single `LoxError` class	Simpler hierarchy, uniform handling	Loses phase information, forces all handlers to check phase type	❌
Phase-specific subclasses	Type-safe, clear intent, enables tailored recovery	More classes, some duplication	✅

Location Tracking Implementation

Both tokens and exceptions must carry precise location information:

Data Structure	Location Fields	Purpose
`Token`	`line: int`, `column: int` (optional)	Record where token appears in source
`ScanError`	`line: int`, `message: String`	Report where scanning failed
`ParseError`	`token: Token`, `message: String`	Report which token caused parse failure
`RuntimeError`	`token: Token`, `message: String`	Report which expression caused runtime failure

Error Recovery Strategies

Mental Model: The Fault-Tolerant Assembly Line Imagine an assembly line that can recover from different types of faults. For minor part defects (lexical errors), workers remove the bad part but keep the line moving. For assembly mistakes (syntax errors), the line pauses, discards the partially assembled unit back to the last known good checkpoint, then resumes. For operational failures (runtime errors), the entire unit is scrapped, but the factory can still produce subsequent units.

Parser: Panic-Mode Recovery

The parser implements **panic-mode recovery**, a technique where upon encountering a syntax error, it discards tokens until reaching a known synchronization point (typically a statement boundary), then continues parsing. This prevents a single syntax error from causing a cascade of spurious follow-up errors.

Panic-Mode Recovery Algorithm:

1. **Detection**: When `Parser.error()` is called with the current token and an error message.
2. **Panic Mode Entry**: Set an internal `panicMode` flag to `true`.
3. **Synchronization Point**: Discard tokens until finding one that can reasonably continue parsing:

- Statement boundaries: `SEMICOLON`, `PRINT`, `RETURN`, `IF`, `WHILE`, `FOR`, `CLASS`, `FUN`, `VAR`
- Class/function boundaries: `RIGHT_BRACE`

4. **Recovery**: Clear `panicMode` flag and continue parsing from the synchronization point.
5. **Error Reporting**: Report the error immediately with token location and expected construct.

> **Design Principle:** The parser should never crash or enter an infinite loop due to invalid syntax. It should report *all* syntax errors in a single pass when possible, giving programmers comprehensive feedback.

ADR: Synchronization Point Selection

> **Decision: Synchronize at Statement Boundaries**
>
> - **Context**: After a syntax error, we need to discard tokens until we reach a point where parsing can reasonably resume without producing garbage.
> - **Options Considered**:
>   1. **Synchronize at statement boundaries** (`;`, keywords starting statements).
>   2. **Synchronize at expression boundaries** (operators, parentheses).
>   3. **Synchronize at any "safe" token** (tokens that can't appear in the current context).
> - **Decision**: Synchronize at statement boundaries.
> - **Rationale**:
>   - **Simplicity**: Statement boundaries are easy to identify and match programmer intuition.
>   - **Effectiveness**: Most syntax errors affect a single statement; discarding the rest of a malformed statement is safe.
>   - **Predictability**: Programmers can understand why parsing resumed where it did.
> - **Consequences**:
>   - May discard multiple tokens for errors early in a statement.
>   - Provides good recovery for most common syntax errors.
>   - Simple to implement with a set of synchronization tokens.

Option	Pros	Cons	Chosen?
Statement boundaries	Simple, effective, predictable	May discard many tokens	✅
Expression boundaries	Finer-grained recovery	Complex to implement, may produce confusing follow-up errors	❌
Contextual "safe" tokens	Most precise recovery	Very complex, requires full context tracking	❌

Scanner: Best-Effort Recovery

The scanner implements a simpler recovery strategy: when encountering an unrecognized character or unterminated string, it reports the error but continues scanning from the next character (for invalid characters) or line (for unterminated strings).

Scanner Recovery Rules:

Error Type	Recovery Action	Rationale
Unrecognized character (e.g., `@`)	Report error, advance past character	Single character likely a typo; rest of file may be valid
Unterminated string	Report error, consume until end of line	Strings can't span lines; assume programmer forgot closing quote
Malformed number (e.g., `123.`)	Report error, tokenize as valid prefix	Helps identify exact location while allowing continued parsing

Interpreter: Fail-Fast Runtime Errors

Runtime errors follow a **fail-fast** strategy: when a semantic violation occurs, execution of the current expression/statement halts immediately, the error propagates up the call stack, and the interpreter stops evaluating the current program (or statement in REPL mode).

Runtime Error Propagation:

1. **Detection**: A runtime check fails in an evaluation method (e.g., type mismatch).
2. **Throw**: `RuntimeError` is thrown with the offending token and descriptive message.
3. **Propagation**: Exception bubbles up through `evaluate()` and `execute()` calls.
4. **Top-Level Catch**: `Interpreter.interpret()` catches `RuntimeError`, reports it, and stops execution of the current program (but keeps interpreter alive in REPL mode).

> **Important Distinction:** Runtime errors halt *execution* but don't crash the *interpreter process*. In batch mode, the interpreter exits with an error code. In REPL mode, it prints the error and returns to the prompt.

Specific Edge Cases and Semantics

Mental Model: The Law of Least Surprise Programming language semantics should follow the "principle of least surprise": edge cases should behave in ways that are intuitive to experienced programmers. When intuition fails, the behavior should be explicitly defined and documented rather than left to implementation chance.

Type System Edge Cases

Lox uses dynamic typing with implicit conversions only where mathematically sensible. The following table defines the complete semantics:

Operation	Left Operand	Right Operand	Result	Error Condition
Addition (`+`)	Number	Number	Numeric sum	None
	String	Any	Concatenation (right converted to string)	None
	Any	String	Concatenation (left converted to string)	None
	Non-string, Non-number	Non-string	`RuntimeError` : "Operands must be two numbers or at least one string."	✅
Other Arithmetic (`-` , `*` , `/`)	Number	Number	Numeric operation	Division by zero → `RuntimeError`
	Non-number	Any	`RuntimeError` : "Operand must be a number."	✅
	Any	Non-number	`RuntimeError` : "Operand must be a number."	✅
Comparison (`<` , `<=` , `>` , `>=`)	Number	Number	Boolean comparison	None
	Non-number	Any	`RuntimeError` : "Operands must be numbers."	✅
	Any	Non-number	`RuntimeError` : "Operands must be numbers."	✅
Equality (`==` , `!=`)	Any	Any	Deep equality comparison	None (even different types can be compared)
Unary Negation (`-`)	Number	-	Negated number	None
	Non-number	-	`RuntimeError` : "Operand must be a number."	✅
Unary Not (`!`)	Any	-	Logical negation (applies truthiness)	None

Truthiness Rules

Lox's truthiness rules are simple but must be implemented consistently:

Value	Truthiness	Rationale
`false`	Falsy	Boolean false
`nil`	Falsy	Represents "no value"
`true`	Truthy	Boolean true
Number (`0` , `0.0` , negative, positive)	Truthy	Common convention: only booleans control flow
String (empty `""` , non-empty)	Truthy	Empty string is still a valid value
Function object	Truthy	Callable entity exists
Class object	Truthy	Type definition exists
Instance object	Truthy	Object exists

> **Key Insight:** Unlike some languages, Lox does *not* treat `0` , `0.0` , or empty string `""` as falsy. This simplifies the mental model: only `false` and `nil` are falsy.

Object Model Edge Cases

Scenario	Behavior	Error Condition
Accessing undefined property	Returns `nil`	None (dynamic languages often allow this)
Setting new property	Creates property on instance	None (objects are open)
Calling non-function value	`RuntimeError` : "Can only call functions and classes."	✅
Calling class as function	Creates new instance, calls `init` if defined	Wrong arity for `init` → `RuntimeError`
`super` outside class	`ParseError` at parse time	✅
`this` outside method	`RuntimeError` : "Can't use 'this' outside of a class."	✅
Return at top level	`RuntimeError` : "Can't return from top-level code."	✅
Inheritance cycle	`RuntimeError` : "Inheritance cycle detected."	✅ (at class definition)

Special Numerical Semantics

Operation	Special Case	Result	Notes
Division (`/`)	`1 / 0`	`RuntimeError` : "Division by zero."	Must be caught at runtime
Negation (`-`)	`-0`	`-0` (negative zero)	IEEE 754 floating-point behavior
String to number	`Number("123")` (hypothetical)	N/A	Lox has no explicit conversion functions

String Concatenation Details

String concatenation follows these conversion rules:

Value Type	String Representation	Example
`nil`	`"nil"`	`"Value: " + nil` → `"Value: nil"`
`false`	`"false"`	`"Status: " + false` → `"Status: false"`
`true`	`"true"`	`"Status: " + true` → `"Status: true"`
Number	Decimal representation (no trailing `.0` for integers)	`"Count: " + 42` → `"Count: 42"`
String	The string itself (no extra quotes)	`"Hello, " + "world"` → `"Hello, world"`
Function	Implementation-defined (e.g., `<fn foo>`)	`"Func: " + foo` → `"Func: <fn foo>"`
Class	Implementation-defined (e.g., `<class Foo>`)	`"Class: " + Foo` → `"Class: <class Foo>"`
Instance	Implementation-defined (e.g., `<Foo instance>`)	`"Obj: " + Foo()` → `"Obj: <Foo instance>"`

Variable Resolution Edge Cases

Scenario	Behavior	Error Condition
Accessing undefined variable	`RuntimeError`: "Undefined variable 'x'."	✅
Assigning to undefined variable	`RuntimeError`: "Undefined variable 'x'."	✅
Redeclaring variable in same scope	Allowed (overwrites previous value)	None (dynamic languages often allow)
Shadowing outer variable	Allowed (inner scope variable masks outer)	None
Closure accessing mutable outer variable	Gets current value at execution time	None (captures variable, not value)

Function Call Edge Cases

Scenario	Behavior	Error Condition
Wrong number of arguments	`RuntimeError`: "Expected X arguments but got Y."	✅
Recursive call depth exceeded	Stack overflow (JVM handles)	None (system-dependent)
Return without value	Returns `nil`	None
Call on `nil`	`RuntimeError`: "Can only call functions and classes."	✅

Inheritance Edge Cases

Scenario	Behavior	Error Condition
Inheriting from non-class	`RuntimeError`: "Superclass must be a class."	✅
Accessing `super` in class with no superclass	`RuntimeError`: "Can't use 'super' in a class with no superclass."	✅
Calling `super.init()` multiple times	Allowed (but unusual)	None
Overriding method with different arity	Allowed (Lox doesn't check)	None

Common Pitfalls

⚠️ Pitfall: Confusing Java null with Lox nil

- **Description**: Using Java's `null` to represent Lox's `nil` without proper boxing.
- **Why it's wrong**: Many operations that should work on `nil` (like concatenation with strings) will throw `NullPointerException` if using raw `null`.
- **Fix**: Create a dedicated `LoxNil` singleton class or use a sentinel object to represent `nil` distinct from Java `null`.

⚠️ Pitfall: Incomplete error location reporting

- **Description**: Reporting errors without line numbers or with incorrect line numbers.
- **Why it's wrong**: Programmers can't find the error in their source code.
- **Fix**: Ensure every `Token` carries accurate line (and ideally column) information, and every error includes the token's location.

⚠️ Pitfall: Forgetting to synchronize after parse error

- **Description**: Parser enters infinite loop or produces cascading errors after first syntax error.
- **Why it's wrong**: Makes interpreter unusable for debugging multi-error programs.
- **Fix**: Implement proper panic-mode recovery with synchronization at statement boundaries.

⚠️ Pitfall: Allowing division by zero to propagate as Java exception

- **Description**: Letting Java's `ArithmeticException` bubble up instead of catching and converting to `RuntimeError`.
- **Why it's wrong**: Gives Java-specific error messages instead of Lox-specific ones.
- **Fix**: Explicitly check divisor before division operation and throw `RuntimeError` with Lox message.

⚠️ Pitfall: Not checking operand types before operations

- **Description**: Assuming operands are numbers without verification, leading to `ClassCastException`.
- **Why it's wrong**: Breaks dynamic typing guarantees and produces Java-level errors.
- **Fix**: Use `instanceof` checks or visitor pattern to verify operand types before operations.

⚠️ Pitfall: Poor truthiness implementation

- **Description**: Treating `0`, `""`, or other values as falsy contrary to Lox spec.
- **Why it's wrong**: Programs behave differently than specified in language definition.
- **Fix**: Implement `isTruthy()` method that returns `false` only for `false` and `nil`.

Implementation Guidance

A. Technology Recommendations

Component	Simple Option	Advanced Option
Error Reporting	Plain `System.err.println()` with formatting	Structured logging framework (SLF4J)
Exception Handling	Java built-in exceptions	Custom exception hierarchy with chained causes
Location Tracking	Line numbers only	Line and column with source snippet display

B. Recommended File/Module Structure

```
lox/
  src/main/java/com/craftinginterpreters/lox/
    Lox.java                # Main entry point with error reporting
    Token.java              # Token definition with location
    TokenType.java          # Enum of token types
    Scanner.java            # Lexical analysis with ScanError
    Parser.java             # Syntactic analysis with ParseError
    Interpreter.java        # Runtime evaluation with RuntimeError
    RuntimeError.java       # Runtime exception class
    ParseError.java         # Parse exception class
    environment/
      Environment.java      # Variable storage
    ast/
      Expr.java             # Expression base class
      Stmt.java             # Statement base class
      ...                   # All AST node classes
    runtime/
      LoxCallable.java      # Callable interface
      LoxFunction.java      # Function representation
      LoxClass.java         # Class representation
      LoxInstance.java      # Instance representation
      LoxNil.java           # Nil singleton
```

C. Error Exception Classes (Complete Starter Code)

JAVA

```java
// RuntimeError.java

package com.craftinginterpreters.lox;


// Runtime errors occur during evaluation when semantic rules are violated

public class RuntimeError extends RuntimeException {

  public final Token token;  // The token where the error occurred



  public RuntimeError(Token token, String message) {

    super(message);

    this.token = token;

  }

}
```

JAVA

```java
// ParseError.java

package com.craftinginterpreters.lox;


// Parse errors occur during parsing when syntax rules are violated

// This is a wrapper exception used for panic-mode recovery

public class ParseError extends RuntimeException {

  public ParseError(Token token, String message) {

    super(message);

    // Note: We could store token for location, but ParseError is caught

    // internally by Parser for recovery, not reported to user directly

  }

}
```

D. Error Reporting in Main Class (Core Logic Skeleton)

```java
// Lox.java (partial)

package com.craftinginterpreters.lox;


public class Lox {

  private static final Interpreter interpreter = new Interpreter();

  static boolean hadError = false;

  static boolean hadRuntimeError = false;



  // Main entry point for running a source file

  public static void runFile(String path) {

    // TODO 1: Read the file contents into a string

    // TODO 2: Call run() with the source string

    // TODO 3: Exit with appropriate code: 65 for syntax errors, 70 for runtime errors

  }



  // Main entry point for REPL

  public static void runPrompt() {

    // TODO 1: Create InputStreamReader for interactive input

    // TODO 2: For each line: run(line), reset hadError flag (but not hadRuntimeError)

    // TODO 3: Handle Ctrl+D gracefully

  }



  // Core execution engine

  public static void run(String source) {

    try {

      // TODO 1: Create Scanner and scan tokens

      // TODO 2: Create Parser and parse statements

      // TODO 3: If no parse error, interpret with Interpreter

    } catch (RuntimeError error) {

      runtimeError(error);

    }

  }



  // Report a scanning or parsing error

  public static void error(int line, String message) {
```

```
    report(line, "", message);
  }

  // Overload for token-based errors
  public static void error(Token token, String message) {
    // TODO 1: Check if token is EOF and report differently
    // TODO 2: Otherwise report with line number and token lexeme
  }

  private static void report(int line, String where, String message) {
    // TODO: Print to stderr: "[line X] ErrorY: message"
    // TODO: Set hadError = true
  }

  // Report a runtime error
  public static void runtimeError(RuntimeError error) {
    // TODO 1: Print to stderr: error.getMessage() + "\n[line " + error.token.line + "]"
    // TODO 2: Set hadRuntimeError = true
  }
}
```

E. Parser Error Recovery (Core Logic Skeleton)

JAVA

```java
// Parser.java (partial)
public class Parser {
  private boolean hadError = false;
  private boolean panicMode = false;

  // Entry point for parsing
  public List<Stmt> parse() {
    List<Stmt> statements = new ArrayList<>();
    while (!isAtEnd()) {
      // TODO 1: Parse a declaration (which parses statements)
      // TODO 2: Add to statements list
    }
    return statements;
  }

  // Report a parse error and enter panic mode
  private ParseError error(Token token, String message) {
    // TODO 1: Call Lox.error(token, message) to report to user
    // TODO 2: Set hadError = true
    // TODO 3: Return new ParseError for synchronization
  }

  // Consume tokens until reaching a synchronization point
  private void synchronize() {
    // TODO 1: Exit panic mode
    // TODO 2: Advance past bad token
    // TODO 3: Skip tokens until reaching a statement boundary:
    //         - SEMICOLON
    //         - Keywords: CLASS, FUN, VAR, FOR, IF, WHILE, PRINT, RETURN
    // TODO 4: Return to normal parsing
  }

  // Helper method used in parsing methods
  private Token consume(TokenType type, String message) {
```

```
    if (check(type)) return advance();

    throw error(peek(), message);  // This triggers panic mode

  }

}
```

F. Runtime Type Checking (Core Logic Skeleton)

```java
// Interpreter.java (partial)
public class Interpreter implements Expr.Visitor<Object>, Stmt.Visitor<Void> {


  // Binary expression evaluation with type checking
  @Override
  public Object visitBinaryExpr(Expr.Binary expr) {
    // TODO 1: Evaluate left and right operands
    // TODO 2: For arithmetic operations (-, *, /):
    //         - Check both operands are Double using checkNumberOperand()
    //         - For division, check right != 0.0
    // TODO 3: For comparison (<, <=, >, >=):
    //         - Check both operands are Double using checkNumberOperands()
    // TODO 4: For equality (==, !=):
    //         - Use isEqual() helper (works for any types)
    // TODO 5: For addition (+):
    //         - If either operand is String, convert both to strings and concatenate
    //         - Otherwise check both are Double
    // TODO 6: Throw RuntimeError with token location for type mismatches
  }


  // Helper to check unary operand type
  private void checkNumberOperand(Token operator, Object operand) {
    // TODO: If operand is not Double, throw RuntimeError
  }


  // Helper to check binary operand types
  private void checkNumberOperands(Token operator, Object left, Object right) {
    // TODO: If either operand is not Double, throw RuntimeError
  }


  // String conversion helper
  private String stringify(Object object) {
    // TODO 1: Handle null (return "nil")
    // TODO 2: Handle Double: remove trailing ".0" for integers
```

```
    // TODO 3: Handle Boolean: return "true" or "false"
    // TODO 4: For other objects, call toString()
  }
}
```

G. Language-Specific Hints (Java)

1. **Use `Double` for numbers**: Java's `double` primitive can't be `null`, so use `Double` object type to represent Lox numbers and allow `nil` distinction.
2. **Create a `LoxNil` singleton**:

```java
public class LoxNil {
  public static final LoxNil INSTANCE = new LoxNil();
  private LoxNil() {}
  @Override public String toString() { return "nil"; }
}
```

3. **Careful with floating-point equality**: Use epsilon comparison for `==` on numbers:

```java
private boolean isEqual(Object a, Object b) {
  if (a == null && b == null) return true;
  if (a == null) return false;
  if (a instanceof Double && b instanceof Double) {
    return Math.abs((Double)a - (Double)b) < 1e-12;
  }
  return a.equals(b);
}
```

4. **Throw, don't return errors**: In visitor methods, throw `RuntimeError` immediately when semantic violation detected rather than returning error sentinel values.

H. Milestone Checkpoint: Error Handling Verification

Milestone	Test Command	Expected Behavior
1 (Scanner)	`java Lox test.lox` with `"unterminated string`	Reports "[line 1] Error: Unterminated string."
2-3 (Parser)	`java Lox test.lox` with `var x = ;`	Reports "[line 1] Error at ';': Expect expression."
4 (Interpreter)	`java Lox test.lox` with `print "hello" - 1;`	Reports "RuntimeError: Operands must be numbers. [line 1]"
5-10 (All)	`java Lox test.lox` with multiple errors	Reports all syntax errors, stops at first runtime error

Debugging Tips Table:

Symptom	Likely Cause	How to Diagnose	Fix
`NullPointerException` in string concatenation	Using Java `null` for Lox `nil`	Check if operand is raw `null` instead of `LoxNil`	Use `LoxNil.INSTANCE` sentinel
Parser goes into infinite loop on syntax error	Missing panic-mode recovery	Add print statements in `synchronize()`	Implement proper statement boundary synchronization
Error reported at wrong line number	Scanner not tracking lines correctly	Test with multi-line string literals	Update line counter when encountering `\n`
Division by zero doesn't throw error	Not checking divisor before operation	Add debug print before division	Explicitly check if divisor equals 0.0
`"hello" + 123` works but `123 + "hello"` doesn't	Only checking left operand type in `+`	Test both operand orders	Check if EITHER operand is string
Closures see wrong variable values	Capturing environment by reference vs value	Print environment IDs during execution	Ensure closures capture environment reference, not copy

8. Testing Strategy

> **Milestone(s):** All milestones (Milestones 1-10) — Testing is a cross-cutting concern that evolves alongside the interpreter, from isolated component validation to integrated system verification.

A robust testing strategy is essential for building a correct and reliable interpreter. The tree-walking interpreter's architecture naturally lends itself to **component-wise testing**—each major phase (scanning, parsing, interpretation) can be tested in isolation before integration. This section outlines a pragmatic testing methodology aligned with the project milestones, providing concrete verification checkpoints to ensure each component works correctly before proceeding to the next.

Mental Model: The Construction Inspector

Think of testing as a **construction inspector** who examines each stage of a building project. First, they check the raw materials (tokens), then the structural framework (AST), then the plumbing and electrical systems (expressions and statements), and finally the complete building with all its features (functions, classes). At each milestone, the inspector has a specific checklist of what should work. This incremental verification prevents foundational defects from propagating upward, where they become exponentially harder to debug.

Unit and Integration Testing

The interpreter's pipeline architecture suggests a natural testing hierarchy: **unit tests** for individual components, **integration tests** for component boundaries, and **end-to-end tests** for complete programs. This layered approach mirrors the development progression through milestones.

Component Isolation Strategy: Each major component (Scanner, Parser, Interpreter) should be testable in isolation with minimal dependencies:

- The **Scanner** can be tested by providing source strings and verifying the output token sequence.
- The **Parser** can be tested by feeding it pre-generated tokens and checking the resulting AST structure.
- The **Interpreter** can be tested by constructing AST nodes directly and verifying evaluation results.

Test Infrastructure Table:

Test Level	Purpose	Input	Verification Method	Example
Scanner Unit Tests	Verify tokenization rules	Source code string	Compare actual token list with expected token types, lexemes, literals, and line numbers	`"var x = 42;"` → `[VAR, IDENTIFIER("x"), EQUAL, NUMBER(42), SEMICOLON]`
Parser Unit Tests	Verify AST construction	Token list	Compare generated AST structure with expected AST (using pretty-printing or visitor)	`[NUMBER(1), PLUS, NUMBER(2)]` → `Binary(Literal(1), PLUS, Literal(2))`
Interpreter Unit Tests	Verify expression evaluation	AST nodes	Compare evaluation result with expected runtime value	`Binary(Literal(1), PLUS, Literal(2))` → `3`
Integration Tests	Verify component handoffs	Source code string	Compare program output with expected output (printed results)	`"print 1 + 2;"` → Console shows `"3"`
End-to-End Tests	Verify complete language features	Complete Lox programs	Compare actual vs. expected output and runtime behavior	Test files with `print` statements and expected output

Testing Visitor Pattern Implementations: The Visitor pattern presents a unique testing challenge because operations are distributed across many `visitXxx` methods. Two effective strategies are:

1. **Pretty Printer Tests:** Implement a `PrettyPrinter` visitor that converts AST nodes back to a canonical string representation (like S-expressions). This provides a language-independent way to verify parser output without requiring a working interpreter.
2. **Evaluation Assertion Tests:** For the interpreter itself, write tests that construct AST nodes programmatically, evaluate them, and assert on the resulting runtime values.

> **Design Insight:** Testing the parser with a pretty printer creates a **self-verifying loop**: source code → tokens → AST → pretty-printed string → compare with expected formatted output. This isolates parser correctness from interpreter implementation details.

Integration Testing Strategy: As components become ready, write tests that exercise the full pipeline:

```
Source Code → Scanner → Parser → Interpreter → Output
```
PLAINTEXT

These tests should verify not just successful execution but also proper error propagation (lexical errors from scanner, syntax errors from parser, runtime errors from interpreter).

Milestone Checkpoints (Verification Guide)

The following table provides concrete verification steps for each milestone. These checkpoints serve as **progress indicators**—if you can successfully run these tests, your implementation is likely correct for that milestone's requirements.

Milestone Verification Table:

Milestone	Test Focus	Sample Test Command (Conceptual)	Expected Output/Behavior	What It Verifies
1. Scanner	Tokenization	Provide `"var x = 42;"` to Scanner	Token sequence: `VAR, IDENTIFIER("x"), EQUAL, NUMBER(42), SEMICOLON`	Keywords, identifiers, numbers, operators, line tracking
	String literals	`"\"hello\""`	`STRING("hello")` with correct literal value	String scanning with quotes, escape sequence handling
	Error reporting	`"@\nvar"`	Scanner reports error at line 1: "Unexpected character '@'"	Lexical error detection with line numbers
2. AST & 3. Parser	Expression parsing	Parse `"1 + 2 * 3"`	AST: `Binary(Literal(1), PLUS, Binary(Literal(2), STAR, Literal(3)))`	Operator precedence (* before +)
	Parentheses	Parse `"(1 + 2) * 3"`	AST: `Binary(Binary(Literal(1), PLUS, Literal(2)), STAR, Literal(3))`	Grouping overrides precedence
	Error recovery	Parse `"1 + ; print 2;"`	Reports syntax error at `;` but continues to parse `print 2;`	Panic-mode recovery to next statement
4. Evaluating Expressions	Arithmetic	Evaluate `"1 + 2 * 3"`	Result: `7` (number)	Binary operator evaluation with precedence
	Type errors	Evaluate `"\"hello\" - 1"`	Runtime error: "Operands must be numbers."	Runtime type checking
	Truthiness	Evaluate `"!false"`	Result: `true`	Boolean logic and truthiness rules
5. Statements and State	Variables	Execute `"var x = 1; print x;"`	Prints `"1"`	Variable declaration, initialization, and lookup
	Assignment	Execute `"var x = 1; x = 2; print x;"`	Prints `"2"`	Variable reassignment
	Scoping	Execute `"{ var x = 1; } print x;"`	Runtime error: "Undefined variable 'x'."	Block scope isolation
6. Control Flow	If/else	Execute `"if (true) print \"yes\"; else print \"no\";"`	Prints `"yes"`	Conditional branching
	While loop	Execute `"var i = 0; while (i < 3) { print i; i = i + 1; }"`	Prints `0`, `1`, `2`	Loop execution with condition
	Short-circuit	Execute `"false and print \"skipped\";"`	Nothing printed (no error)	Short-circuit evaluation of `and`
7. Functions	Function call	Execute `"fun sayHi() { print \"Hi!\"; } sayHi();"`	Prints `"Hi!"`	Function declaration and invocation

Milestone	Test Focus	Sample Test Command (Conceptual)	Expected Output/Behavior	What It Verifies
	Parameters	Execute `"fun add(a, b) { return a + b; } print add(1, 2);"`	Prints `"3"`	Parameter binding and return values
	Recursion	Execute `"fun fib(n) { if (n <= 1) return n; return fib(n-1) + fib(n-2); } print fib(5);"`	Prints `"5"`	Recursive function calls with proper environments
8. Closures	Closure capture	Execute `"fun makeCounter() { var i = 0; fun count() { i = i + 1; return i; } return count; } var counter = makeCounter(); print counter(); print counter();"`	Prints `1`, then `2`	Functions capture enclosing environment
	Multiple closures	Execute closure test with two independent counters	Each counter maintains separate state	Each closure captures its own environment reference
9. Classes	Instance creation	Execute `"class Point {} var p = Point(); print p;"`	Prints instance representation (e.g., `"Point instance"`)	Class as constructor, instance creation
	Fields and methods	Execute class with `init` and method	Instance fields initialized, methods bound with `this`	Property access, method binding, initializer
10. Inheritance	Method inheritance	Execute subclass that inherits method from superclass	Subclass instance can call inherited method	Single inheritance, method resolution
	Super calls	Execute subclass method that calls `super.method()`	Superclass version of method executes	`super` keyword resolution and invocation

Verification Procedure: For each milestone, follow this step-by-step verification process:

1. **Isolated Component Tests:** First, test the new component in isolation using unit tests.
2. **Integration with Previous Components:** Test the new component integrated with all previously implemented components.
3. **Edge Case Validation:** Test boundary conditions and error cases specific to the milestone.
4. **Regression Testing:** Ensure existing functionality from previous milestones still works correctly.

> **Key Principle: Test incrementally and verify often.** Don't wait until the entire interpreter is built to test it. Each milestone's acceptance criteria provide specific behaviors to test.

Common Pitfalls in Testing

⚠️ Pitfall: Testing Only Happy Paths

- **Description:** Writing tests that only verify correct programs execute successfully, neglecting error cases and edge conditions.
- **Why It's Wrong:** An interpreter must handle malformed programs gracefully with clear error messages. Missing error tests leaves critical functionality unverified.
- **Fix:** For each feature, write tests for: (1) valid usage, (2) type errors, (3) undefined variables, (4) syntax errors, (5) boundary conditions (empty strings, zero, nil).

⚠️ Pitfall: Over-Reliance on End-to-End Tests

- **Description:** Testing everything through complete program execution, making it difficult to pinpoint which component has a bug.
- **Why It's Wrong:** When a test fails, you must debug the entire pipeline rather than an isolated component.

- **Fix:** Maintain a balanced test pyramid: many unit tests, fewer integration tests, minimal end-to-end tests. Use unit tests to verify component logic in isolation.

⚠️ Pitfall: Hard-Coding Test Values

- **Description:** Writing tests that depend on specific string representations (e.g., `"LoxFunction@4f023edb"`) that vary across runs.
- **Why It's Wrong:** Such tests are fragile and may fail due to unrelated changes like JVM memory address variations.
- **Fix:** Test semantic behavior, not string representations. For functions and classes, test that they can be called, not how they convert to strings.

⚠️ Pitfall: Neglecting Line Number Tracking

- **Description:** Forgetting to test that error messages include accurate line and column numbers.
- **Why It's Wrong:** Without accurate location information, debugging user programs becomes difficult.
- **Fix:** In error test cases, explicitly verify that reported line numbers match the source location.

Architecture Decision Record: Test-First vs. Test-After Development

> **Decision: Test-After Development with Milestone Checkpoints**

Context: We need a testing strategy suitable for learners building an interpreter incrementally. The interpreter has clear component boundaries and well-defined milestones, each adding specific functionality.

Options Considered:

1. **Test-Driven Development (TDD):** Write tests before implementing each feature.
2. **Test-After with Milestone Verification:** Implement functionality first, then write tests to verify the milestone's acceptance criteria.
3. **Ad-hoc Manual Testing:** Manually test features as implemented without systematic test suites.

Decision: We adopt **Test-After with Milestone Verification**. After completing each milestone's implementation, learners write comprehensive tests to verify all acceptance criteria.

Rationale:

- **Learning Focus:** For educational purposes, understanding how to implement features precedes learning test-driven methodologies.
- **Clear Verification:** Milestone checkpoints provide concrete, testable outcomes that learners can verify systematically.
- **Progressive Complexity:** Early milestones (Scanner, Parser) benefit from unit testing; later milestones (Interpreter features) benefit from integration testing.
- **Practical Reality:** Learners often need to see the implementation working before they understand what to test.

Consequences:

- **Positive:** Learners gain experience writing tests that verify specific behaviors. The milestone structure provides natural test boundaries.
- **Negative:** Some implementation bugs might only be caught after implementation is complete. Requires discipline to write tests for all acceptance criteria.
- **Mitigation:** The verification guide provides explicit test cases for each milestone, ensuring comprehensive coverage.

Comparison Table:

Option	Pros	Cons	Suitability for Learning
Test-Driven Development	Ensures testable design, comprehensive test coverage from start	Requires understanding of feature behavior before implementation, can feel artificial for learners	Low: Adds cognitive load on top of implementation challenges
Test-After with Verification	Natural progression: implement then verify, milestone checkpoints guide testing	May miss edge cases without upfront test design, requires retroactive test writing	High: Matches how learners naturally build (implement then test)
Ad-hoc Manual Testing	Quick feedback, no test maintenance overhead	Inconsistent coverage, difficult to regression test, errors may slip through	Low: Doesn't teach systematic testing practices

Implementation Guidance

While the main design avoids code, this implementation guidance provides practical testing infrastructure to verify each milestone.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Testing Framework	JUnit 5 (standard Java testing)	TestNG (more features)
Assertion Library	JUnit's built-in assertions	AssertJ (fluent assertions)
Test Organization	One test class per component (ScannerTest, ParserTest, etc.)	Nested test classes by feature
Test Data Management	Hard-coded strings in test methods	External test files for programs
Mocking	Manual test doubles	Mockito (for complex dependencies)

B. Recommended File/Module Structure

```
lox-interpreter/
  src/main/java/com/craftinginterpreters/lox/
    Lox.java                    # Main entry point
    Scanner.java                # Milestone 1
    Token.java                  # Milestone 1
    TokenType.java              # Milestone 1
    parser/
      Parser.java               # Milestone 3
      ParseError.java           # Milestone 3
    ast/
      Expr.java                 # Milestone 2
      Stmt.java                 # Milestone 2
      AstPrinter.java           # Milestone 2 (for testing)
    interpreter/
      Interpreter.java          # Milestone 4+
      Environment.java          # Milestone 5
      RuntimeError.java         # Milestone 4
      LoxFunction.java          # Milestone 7
      LoxClass.java             # Milestone 9
      LoxInstance.java          # Milestone 9
  src/test/java/com/craftinginterpreters/lox/
    ScannerTest.java            # Tests for Milestone 1
    ParserTest.java             # Tests for Milestones 2-3
    InterpreterTest.java        # Tests for Milestones 4-10
    testdata/                   # Directory for test programs
      hello.lox
      arithmetic.lox
      functions.lox
      classes.lox
```

C. Infrastructure Starter Code: Test Utilities

For comprehensive testing, implement these utility classes:

JAVA

```java
// TestUtilities.java - COMPLETE utility methods for testing

package com.craftinginterpreters.lox;


import java.util.List;

import java.util.ArrayList;


public class TestUtilities {

    // Helper to create token list for parser tests

    public static List<Token> tokenList(Token... tokens) {

        return new ArrayList<>(List.of(tokens));

    }



    // Helper to create a NUMBER token

    public static Token numberToken(double value, int line) {

        return new Token(TokenType.NUMBER, String.valueOf(value), value, line);

    }



    // Helper to create a STRING token

    public static Token stringToken(String value, int line) {

        return new Token(TokenType.STRING, "\"" + value + "\"", value, line);

    }



    // Helper to create an IDENTIFIER token

    public static Token identifierToken(String name, int line) {

        return new Token(TokenType.IDENTIFIER, name, null, line);

    }



    // Helper to compare two values with tolerance for floating point

    public static boolean valuesEqual(Object a, Object b) {

        if (a == null && b == null) return true;

        if (a == null) return false;



        if (a instanceof Double && b instanceof Double) {

            return Math.abs((Double)a - (Double)b) < 0.000001;

        }

        return a.equals(b);
```

```
    }

}
```

D. Core Test Skeleton Code

Scanner Test Skeleton:

JAVA

```java
// ScannerTest.java - TODOs for core scanner tests

package com.craftinginterpreters.lox;


import org.junit.jupiter.api.Test;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;


class ScannerTest {

    @Test

    void testScanSingleTokens() {

        // TODO 1: Create scanner with source: "(){},.-+;/*"

        // TODO 2: Call scanTokens()

        // TODO 3: Verify token types: LEFT_PAREN, RIGHT_PAREN, LEFT_BRACE,

        //         RIGHT_BRACE, COMMA, DOT, MINUS, PLUS, SEMICOLON, SLASH, STAR

        // TODO 4: Verify no errors reported

    }



    @Test

    void testScanKeywords() {

        // TODO 1: Create scanner with source: "var fun class if else while for return"

        // TODO 2: Call scanTokens()

        // TODO 3: Verify token types: VAR, FUN, CLASS, IF, ELSE, WHILE, FOR, RETURN

        // TODO 4: Verify lexemes match keywords exactly

    }



    @Test

    void testScanNumbers() {

        // TODO 1: Create scanner with source: "123 45.67 .5" (note: .5 is invalid)

        // TODO 2: Call scanTokens()

        // TODO 3: Verify NUMBER tokens with correct literal values (123, 45.67)

        // TODO 4: Verify error reported for ".5" (or tokenized as DOT then 5)

    }



    @Test

    void testScanStrings() {

        // TODO 1: Create scanner with source: "\"hello\" \"world\\n\""
```

```
        // TODO 2: Call scanTokens()

        // TODO 3: Verify STRING tokens with values "hello" and "world\n"

        // TODO 4: Verify line counting within strings (escaped newline doesn't increment)

    }


    @Test

    void testScanErrors() {

        // TODO 1: Create scanner with source: "@ # $\nvar"

        // TODO 2: Call scanTokens()

        // TODO 3: Verify errors reported for '@', '#', '$' at line 1

        // TODO 4: Verify VAR token still scanned after errors

    }

}
```

Parser Test Skeleton:

JAVA

```java
// ParserTest.java - TODOs for core parser tests

package com.craftinginterpreters.lox;

import org.junit.jupiter.api.Test;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

class ParserTest {

    @Test

    void testParseArithmetic() {

        // TODO 1: Create tokens: 1 + 2 * 3

        // TODO 2: Create parser with tokens

        // TODO 3: Call parse() to get statements

        // TODO 4: Use AstPrinter to convert AST to string

        // TODO 5: Verify string matches expected: "(+ 1 (* 2 3))" or similar

    }



    @Test

    void testParseParentheses() {

        // TODO 1: Create tokens: (1 + 2) * 3

        // TODO 2: Parse and get AST

        // TODO 3: Verify structure: (* (+ 1 2) 3)

    }



    @Test

    void testParseComparison() {

        // TODO 1: Create tokens: 1 < 2 == true

        // TODO 2: Parse and get AST

        // TODO 3: Verify structure: (== (< 1 2) true) with correct precedence

    }



    @Test

    void testParseErrorRecovery() {

        // TODO 1: Create tokens with error: 1 + ; print 2;

        // TODO 2: Parse - should report error at SEMICOLON

        // TODO 3: Verify parser continues and produces AST for "print 2;"
```

```
    }

}
```

Interpreter Test Skeleton:

JAVA

```java
// InterpreterTest.java - TODOs for core interpreter tests

package com.craftinginterpreters.lox;

import org.junit.jupiter.api.Test;

import java.io.ByteArrayOutputStream;

import java.io.PrintStream;

import static org.junit.jupiter.api.Assertions.*;

class InterpreterTest {

    @Test

    void testEvaluateArithmetic() {

        // TODO 1: Create AST: Binary(Literal(1.0), PLUS, Literal(2.0))

        // TODO 2: Create interpreter

        // TODO 3: Call evaluate() on AST

        // TODO 4: Verify result equals 3.0 (with floating tolerance)

    }



    @Test

    void testRuntimeTypeError() {

        // TODO 1: Create AST: Binary(Literal("hello"), MINUS, Literal(1))

        // TODO 2: Create interpreter

        // TODO 3: Call evaluate() - should throw RuntimeError

        // TODO 4: Verify error message: "Operands must be numbers."

    }



    @Test

    void testVariableScope() {

        // TODO 1: Create AST for: { var x = 1; print x; }

        // TODO 2: Create interpreter

        // TODO 3: Call interpret() with statements

        // TODO 4: Capture output, verify prints "1"

        // TODO 5: Test nested scope: { var x = 1; { var x = 2; } print x; }

    }



    @Test

    void testFunctionCall() {
```

```
        // TODO 1: Create AST for function declaration and call
        // TODO 2: Interpret statements
        // TODO 3: Verify function is callable and returns correct value
        // TODO 4: Test recursion with factorial function
    }

    @Test
    void testClassInheritance() {
        // TODO 1: Create AST for class hierarchy: class A { method() { return 1; } } class B < A {}
        // TODO 2: Interpret statements
        // TODO 3: Create instance of B, call method()
        // TODO 4: Verify returns 1 (inherited method)
    }
}
```

E. Language-Specific Hints (Java)

1. **Capturing Console Output:** Use `ByteArrayOutputStream` and `PrintStream` to capture `System.out` for testing print statements:

```java
ByteArrayOutputStream output = new ByteArrayOutputStream();
System.setOut(new PrintStream(output));
// Run interpreter
assertEquals("3\n", output.toString());
```

2. **Testing Exceptions:** Use JUnit's `assertThrows` to verify errors:

```java
RuntimeError error = assertThrows(RuntimeError.class, () -> interpreter.evaluate(ast));
assertTrue(error.getMessage().contains("Operands must be numbers"));
```

3. **Floating Point Comparisons:** Use epsilon comparison for double values:

```java
private static final double EPSILON = 0.000001;
assertTrue(Math.abs((Double)result - expected) < EPSILON);
```

4. **Visitor Pattern Testing:** When testing visitors, you can create anonymous visitor implementations for specific test cases rather than relying on the full interpreter.

F. Milestone Checkpoint Commands

For each milestone, after implementing the feature, run these verification commands:

Milestone 1 (Scanner):

```bash
# Run scanner tests
./gradlew test --tests "*.ScannerTest"

# Expected: All tests pass, with green output showing tokenization works
# If tests fail, check error messages to see which tokens are incorrect
```

Milestones 2-3 (Parser & AST):

```bash
# Run parser tests
./gradlew test --tests "*.ParserTest"

# Manually test with a simple expression
java -cp build/classes/java/main com.craftinginterpreters.lox.Lox <<< "print (1 + 2) * 3;"
# Should parse successfully (may not evaluate yet)
```

Milestone 4 (Evaluating Expressions):

```bash
# Run interpreter expression tests
./gradlew test --tests "*.InterpreterTest.testEvaluate*"

# Test manually
java -cp build/classes/java/main com.craftinginterpreters.lox.Lox <<< "print 1 + 2 * 3;"
# Should print: 7
```

Milestone 5 (Statements and State):

```bash
# Test variable functionality
java -cp build/classes/java/main com.craftinginterpreters.lox.Lox <<< "var x = 1; x = x + 2; print x;"
# Should print: 3
```

Milestone 6 (Control Flow):

```bash
# Test if and while
java -cp build/classes/java/main com.craftinginterpreters.lox.Lox <<< "
var i = 0;
while (i < 3) {
  if (i % 2 == 0) {
    print \"even: \" + i;
  }
  i = i + 1;
}"
# Should print: even: 0, even: 2
```

Milestone 7 (Functions):

```bash
# Test function declaration and call
java -cp build/classes/java/main com.craftinginterpreters.lox.Lox <<< "
fun factorial(n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}
print factorial(5);"
# Should print: 120
```

Milestone 8 (Closures):

```bash
# Test closure capture
java -cp build/classes/java/main com.craftinginterpreters.lox.Lox <<< "
fun makeCounter() {
  var i = 0;
  fun count() {
    i = i + 1;
    return i;
  }
  return count;
}
var c = makeCounter();
print c();
print c();"
# Should print: 1, then 2
```

Milestone 9 (Classes):

```bash
# Test class instantiation
java -cp build/classes/java/main com.craftinginterpreters.lox.Lox <<< "
class Point {
  init(x, y) {
    this.x = x;
    this.y = y;
  }
}
var p = Point(1, 2);
print p.x;"
# Should print: 1
```

Milestone 10 (Inheritance):

```bash
# Test inheritance chain
java -cp build/classes/java/main com.craftinginterpreters.lox.Lox <<< "
class A {
  method() { return \"A\"; }
}
class B < A {
  method() { return super.method() + \"B\"; }
}
var b = B();
print b.method();"
# Should print: AB
```

G. Debugging Tips for Test Failures

Symptom	Likely Cause	How to Diagnose	Fix
Scanner tests fail on multi-character operators	Not advancing `current` past both characters	Add debug prints showing which characters are being examined	Ensure `match()` advances position when successful
Parser goes into infinite loop	Left-recursive grammar rule	Check that parsing methods always consume at least one token before recursing	Rewrite grammar to eliminate left recursion or add progress check
AST structure incorrect	Wrong precedence or associativity	Use AstPrinter to visualize AST, compare with expected	Verify parsing method order matches precedence table
Variable lookup finds wrong value	Environment chain traversal incorrect	Print environment chain during lookup, check parent links	Ensure `enclosing` field set correctly when creating nested environments
Closure captures wrong environment	Capturing call environment instead of definition environment	Check which environment is stored in LoxFunction	Capture defining environment when function is created, not when called
Method call returns nil unexpectedly	Return statement not propagating value	Add debug to trace return exception throwing/catching	Ensure `visitReturnStmt` throws exception caught in `call()` method
Super calls wrong method	Incorrect method resolution order	Print class hierarchy during super lookup	Walk from current class's superclass, not instance's class

Effective Debugging Techniques:

1. **Add AST Visualization:** Implement a `toString()` method for each AST node or use the AstPrinter to see the actual structure.
2. **Environment Dumper:** Add a method to print all variables in an environment chain.
3. **Execution Tracing:** Add flags to log each evaluation step: `TRACE=true` environment that prints every visit method call.
4. **Unit Test Isolation:** When a test fails, create a minimal reproducing test case to isolate the issue.
5. **Use a Debugger:** Set breakpoints in key methods like `visitBinaryExpr`, `Environment.get`, `LoxFunction.call`.

> **Testing Philosophy:** The goal of testing in this educational project is not just to verify correctness, but to **build understanding**. Each test you write forces you to think through edge cases and semantics, deepening your comprehension of the language you're implementing.

9. Debugging Guide

> **Milestone(s):** All milestones (Milestones 1-10) — Debugging is an integral skill for any software project, but for an interpreter, it involves reasoning across multiple layers of abstraction. This guide provides a structured reference for diagnosing and fixing common issues that arise during implementation, categorized by observable symptoms. It also suggests practical techniques to illuminate the interpreter's internal state.

Interpreting a programming language involves a complex dance between static structures (tokens, AST) and dynamic behavior (environments, function calls). When a bug surfaces, it can be challenging to pinpoint which layer is at fault. This debugging guide is designed to help you develop a systematic approach: start from the observable symptom (e.g., a crash, incorrect output, or infinite loop), trace it back through the data flow, and identify the faulty component or logic.

The guide is structured as a two-part toolkit. First, a comprehensive table maps specific symptoms to their likely root causes and provides targeted fixes. Second, it recommends proactive debugging techniques—like adding visualization tools to your interpreter—that can turn a opaque execution into a transparent, step-by-step narrative.

Symptom → Cause → Fix Table

This table catalogs common issues you may encounter while building your Lox interpreter. For each symptom, we describe the typical manifestation, the underlying cause in the design or implementation, and a concrete fix to apply.

Symptom (What you observe)	Likely Cause (Where the bug lives)	Steps to Diagnose	Recommended Fix
Parser goes into an infinite loop or stack overflows on a simple expression like `1 + 2`.	**Left-recursive grammar rule.** In your parsing method for a precedence level (e.g., `expression()`), you might be unconditionally calling itself first, creating infinite recursion. For example, `expression() -> equality() -> comparison() -> term() -> factor() -> unary() -> primary() -> expression()` forms a cycle.	Add print statements at the start of each parsing method (`expression()`, `equality()`, etc.) to see the call sequence. You'll see the same method called repeatedly without consuming tokens.	Ensure your parsing functions follow the pattern: parse a lower-precedence operator, then loop to parse higher-precedence operators. For example, `expression()` should call `equality()`, then loop while seeing `OR` tokens. The base of the recursion must be `primary()`, which consumes a literal or parenthesized expression without recursively calling the top-level `expression()`.
`"Unterminated string"` error is reported for a perfectly valid string like `"hello"`.	**Incorrect handling of the closing quote.** The scanner's string literal logic advances until it finds a `"` but doesn't check for the end of the source file (`isAtEnd()`). If the string is the last token in the file, it may run past the end and report an error.	Check the scanner's `string()` method. Does it break only when it sees `"`? What happens when `isAtEnd()` becomes true first?	In the `while` loop that scans string characters, add a check for `isAtEnd()`. If reached, report an error: "Unterminated string." and break. Ensure the loop condition is `while (!isAtEnd() && peek() != '"')`.
Variable lookup finds the wrong value or says `"Undefined variable"` for a variable that should be in scope.	**Incorrect environment chain traversal.** The `Environment.get()` method searches only the immediate scope, not the parent chain. Or, `Environment.assign()` updates the wrong scope, modifying a parent's binding when it should shadow.	Print the environment chain during lookup. For a variable `x`, dump each environment's `values` map along the `enclosing` links. See if the binding exists in a different scope than expected.	Ensure `Environment.get(Token name)` recursively checks the current scope, then `enclosing` (if not `null`). `Environment.assign(Token name, Object value)` must follow the same search path and update the *first* environment where the variable is found (or error if none).
`print 1 + "string";` crashes or produces gibberish instead of a runtime error.	**Missing runtime type checking.** The `visitBinaryExpr` method for the `PLUS` operator directly performs arithmetic or concatenation without verifying the operands' types.	Check the code handling the `PLUS` token. Is there a call to `checkNumberOperands()` or similar validation? Does it handle the string concatenation case separately?	Implement `checkNumberOperands()` and call it for arithmetic operators (`MINUS`, `SLASH`, `STAR`). For `PLUS`, explicitly check: if both operands are numbers, add; if both are strings, concatenate; else, throw a `RuntimeError`.
Logical operator `and` does not short-circuit; it always evaluates both sides.	**Eager evaluation of operands.** In `visitLogicalExpr`, you likely evaluate `left` and `right` unconditionally before checking the operator.	Add a print statement before evaluating each operand. You'll see both prints even when the left operand of an `and` is false.	Evaluate the left operand first. For `AND`: if the left is falsy, return it immediately. For `OR`: if the left is truthy, return it immediately. Only evaluate the right operand if needed.
Function calls work once but incorrectly share local variables	**Reusing the same environment for each call.** The	Print the environment's identity (e.g., `System.identityHashCode(environment)`)	In `LoxFunction.call()`, create a *new* `Environment` for each call, setting the closure (the function's

Symptom (What you observe)	Likely Cause (Where the bug lives)	Steps to Diagnose	Recommended Fix
between subsequent calls.	`LoxFunction.call()` method might be using a single environment instance for the function's local scope, rather than creating a new one per invocation.	inside the function body for two calls. If it's the same, variables are being shared.	`closure` field) as its parent. Bind parameters in this fresh environment.
Closure captures the *value* of an outer variable at definition time, not a *reference*; modifying the outer variable doesn't affect the closure.	**Environment capture by copy, not by reference.** The closure might be storing a *copy* of the defining environment's variable map, rather than a reference to the environment object itself.	Check how the closure is created. Is it storing a new `Environment` object with the same mappings, or is it storing a reference to the existing environment?	Ensure the `LoxFunction` stores a reference to the *defining* `Environment` object (the one active when the function was declared). Do not copy its `values` map.
`this` **inside a method refers to the class, not the instance, or throws "Undefined variable 'this'."**	**Missing `this` binding in the method's environment.** When a method is called, the interpreter must bind `this` in the method's local environment to the instance on which the method was invoked.	In `visitCallExpr`, check how you prepare the environment for a method call. Is `this` defined? For a `LoxFunction` that represents a method, does its `call()` method bind `this`?	In `LoxFunction.bind(LoxInstance instance)`, create a new environment with the closure as parent, define `"this"` as the instance, and return a new `LoxFunction` that uses this environment. In `visitGetExpr`, if the property is a method, call `bind(instance)` on it.
`super.method()` **calls the subclass's overridden method, not the superclass version.**	**Incorrect resolution of `super`.** The `visitSuperExpr` might be looking up the method starting from the instance's class, which would find the overridden method first.	Print the class hierarchy in `visitSuperExpr`. Are you correctly accessing the superclass from the current class? Is the method lookup starting from the superclass?	In `visitSuperExpr`, resolve the superclass via the interpreter's `environment` (or a dedicated `super` environment). Then, call `findMethod` on the *superclass* object, not the current class.
Division by zero (e.g., `1 / 0`**) crashes with an arithmetic exception instead of a Lox runtime error.**	**Native Java arithmetic exception is not caught.** The interpreter uses Java's `/` operator, which throws `ArithmeticException` for integer division by zero.	Check if the division operation in `visitBinaryExpr` is wrapped in a try-catch for Java's `ArithmeticException`.	Before performing division, explicitly check if the right operand is zero (or within an epsilon for floating-point). If so, throw a Lox `RuntimeError`. Alternatively, catch Java's `ArithmeticException` and convert it.
The parser reports "Expect expression" when encountering a valid expression after an error.	**Poor error recovery leaves the parser in a bad state.** After a parse error, the parser might not synchronize to a known boundary (like a semicolon), causing subsequent valid code to be misparsed.	Introduce a syntax error early in a multi-statement program. Observe if later statements are parsed correctly.	Implement panic-mode synchronization in `Parser`. In `synchronize()`, discard tokens until you reach a statement boundary (e.g., `SEMICOLON`, `CLASS`, `FUN`, `VAR`, `FOR`, `IF`, `WHILE`, `PRINT`, `RETURN`).
A loop `while (true) { }` **hangs the**	**No timeout or interrupt mechanism.** The tree-walking interpreter will obediently execute the loop	This is expected behavior for a correct infinite loop, but it can be problematic during testing.	For development, consider adding an optional execution step limit or a timeout. Increment a counter in

Symptom (What you observe)	Likely Cause (Where the bug lives)	Steps to Diagnose	Recommended Fix
interpreter indefinitely with no way to stop.	forever, as there's no built-in limit.		`visitWhileStmt` and throw a runtime error after a large number (e.g., 1,000,000) iterations.
`nil` appears in arithmetic operations without a runtime error (e.g., `nil + 1` yields something).	**Missing null/nil checks in type validation.** The `checkNumberOperand` or `checkNumberOperands` functions might only check for `Double` type, not rejecting `nil`.	Test `nil + 1`. Does it throw? Check `isTruthy` and `isEqual` functions as well—they must handle `nil` explicitly.	In `checkNumberOperand` and `checkNumberOperands`, explicitly verify the operand is an instance of `Double` (or `LoxNumber`). If it's `nil` or any other non-number, throw a `RuntimeError`.
Multiline string literals are not allowed, or newlines inside strings are lost.	**Scanner's string literal logic stops at newline characters.** The scanner might treat `\n` as a whitespace character to be skipped, rather than as part of the string.	Check the `string()` method. Does it break when it sees `\n`? Does it handle escape sequences like `\n`?	In the string scanning loop, allow newline characters (`peek() == '\n'`) and increment the line counter. To support escape sequences, when you see a backslash, call a helper to translate the escape sequence (e.g., `\n` to newline character).
`print` statement outputs `null` for `nil` instead of the word `"nil"`.	**The `stringify()` method returns Java's `null` string representation.** When the value is `LoxNil` or Java `null`, `String.valueOf(value)` returns `"null"`.	Examine `Interpreter.stringify(Object object)`. How does it handle the `nil` value?	Explicitly check for `nil` in `stringify()`. If the object is `nil` (or `instanceof LoxNil`), return the string `"nil"`. For booleans, return `"true"` or `"false"` instead of `"true"` / `"false"` as Java strings.
Class inheritance creates a cycle (class A < B and class B < A) and crashes.	**No cycle detection in class inheritance.** The parser or interpreter might allow a class to inherit from itself, directly or indirectly, leading to infinite recursion in method lookup.	Try defining two classes that inherit from each other. Does the parser accept it? What happens at runtime?	In `visitClassStmt`, after resolving the superclass expression, traverse the superclass chain starting from the current class. If you encounter the class being defined, report an error. Store a visited set or simply check that the superclass is not the class itself (direct cycle).
Return statement inside a nested block (but outside a function) causes a confusing error or crash.	**The return exception is not caught at the appropriate level.** The `Return` exception (used to unwind the call stack) might be caught only at the top level, or not caught at all, causing program termination.	Place a `return` statement outside any function. Does the interpreter report a clear error?	Ensure `visitReturnStmt` throws a custom `Return` exception (containing the value). In `executeBlock()` and function call execution, catch this exception only when you are within a function context. If caught at the wrong level, rethrow it. At the top level, if a `Return` exception escapes, report an error: "Cannot return from top-level code."
`fun` keyword used as a variable name is incorrectly tokenized as a keyword, causing a parse error.	**Scanner's keyword detection does not check for identifiers after the keyword.** The scanner might match `fun` as the `FUN` keyword even when it's part	The scanner should match the longest possible lexeme. For `fun`, it's a keyword only if the following character is not a letter or digit (i.e., it's a separate token).	In the scanner, after identifying an alphabetic character (start of an identifier/keyword), consume while `isAlphaNumeric(peek())`. Then, check the resulting lexeme against the keyword map. This ensures `fun` is recognized only as a complete token.

Symptom (What you observe)	Likely Cause (Where the bug lives)	Steps to Diagnose	Recommended Fix
	of a longer identifier like `function`.		
The interpreter incorrectly allows reassignment to a variable that was declared with `var` in a deeper scope but not in the current scope.	**`Environment.assign()` searches and updates the first matching variable in the chain, which may be in an outer scope, incorrectly modifying a different variable than intended.**	This is actually the correct behavior for lexical scoping: assignment should modify the nearest enclosing variable with that name. If you want to prevent this (some languages do), you need a different rule.	If the desired semantics are that assignment can only modify variables declared in the current scope, then `assign()` should not traverse the parent chain. However, Lox uses lexical scoping for assignment, so the described behavior is correct. Verify your understanding of the language spec.
Using `print` as a variable name causes a conflict with the `print` statement.	**The scanner tokenizes `print` as the `PRINT` keyword regardless of context.** In Lox, `print` is a reserved keyword and cannot be used as an identifier.	Check the language grammar: `print` is a keyword for the print statement, not a built-in function. Thus, it is reserved.	This is by design. The scanner should recognize `print` as a `PRINT` token. The parser will then expect it as the start of a print statement. If you want to use `print` as a variable name, you would need to change the language, but that's outside the spec.

Effective Debugging Techniques

Beyond reactive troubleshooting, you can embed diagnostic tools directly into your interpreter to shed light on its internal operations. Think of these techniques as installing surveillance cameras in a factory: you can watch the raw materials (tokens) move along the conveyor belt, see how they're assembled into products (AST nodes), and observe the final testing (evaluation). When a defect occurs, you can review the footage to pinpoint the malfunctioning station.

1. AST Printer: Visualizing the Syntax Tree

The **AST Printer** is a visitor that converts an AST back into a human-readable string format, often as an S-expression. It's invaluable for verifying that your parser is building the correct tree structure.

> **Mental Model:** The AST Printer is like a cartographer who translates a detailed, three-dimensional terrain model (the AST) back into a two-dimensional map with standardized symbols. By comparing the map to the source code, you can confirm the terrain was surveyed correctly.

How to implement: You already built a basic printer in Milestone 2. Enhance it to be more detailed. For each node type, output a parenthesized representation that includes the node type and its key data (e.g., `(Binary + (Literal 1) (Literal 2))`). Call the printer right after parsing and log the output.

What it reveals: Incorrect operator precedence (e.g., `1 + 2 * 3` parsed as `(+ 1 (* 2 3))` vs. `(+ 1 2) (* 3)`), missing nodes, or misplaced parentheses.

2. Environment Dumper: Mapping the Variable Universe

An **Environment Dumper** is a utility that prints the entire chain of environments, showing all variable bindings at a given point in execution. This is crucial for debugging scope, closure, and variable resolution issues.

> **Mental Model:** The Environment Dumper is like a building superintendent who provides a floor-by-floor directory of all tenants (variables) and their current apartment contents (values). When someone can't find a tenant, the superintendent can check which floor the tenant is registered on.

How to implement: Add a method `dump()` to the `Environment` class that iterates through the `values` map and prints each name-value pair, then recursively calls `dump()` on the `enclosing` environment (with an indentation increase). Call this at key points: before/after a

function call, inside a block, or when a "undefined variable" error occurs.

What it reveals: Whether a variable is defined in the current scope or a parent, if a closure captured the correct environment, and if `this` is bound in a method.

3. Step-by-Step Execution Log: The Interpreter's Diary

An **Execution Log** is a trace of every step the interpreter takes as it walks the AST. It logs each node visited (`visitBinaryExpr` , `visitLiteralExpr` , etc.), the values computed, and any side effects (variable definition, assignment).

> **Mental Model:** The Execution Log is like a flight data recorder, meticulously timestamping every control input, sensor reading, and system state change during a flight. In a crash, the log tells you exactly what happened in the moments before.

How to implement: Add a static logging flag to your interpreter. In each `visit` method, log the method name and relevant details (e.g., `"Evaluating binary + with left=1.0, right=2.0"`). Use indentation to show the depth of recursion. Log environment changes separately.

What it reveals: The order of evaluation, short-circuiting behavior, function call entry/exit, and the exact point where a runtime error is thrown.

4. Token Stream Visualizer: Seeing the Lexical Stream

A **Token Stream Visualizer** simply prints the list of tokens produced by the scanner, one per line, with their type, lexeme, and line number. This helps verify that the scanner is correctly categorizing the source code.

How to implement: After calling `scanTokens()` , iterate through the list and print each token using its `toString()` method. Ensure you include identifiers, literals, and punctuation.

What it reveals: Misclassified tokens (e.g., `=` vs `==`), incorrect handling of whitespace/comments, and malformed number or string literals.

5. Manual Test Harness with Assertions

Create a suite of small, focused Lox programs that test a single feature (e.g., variable shadowing, closure capture, super call). In each test, use `print` statements as assertions. For example, a test for closure capture might be:

```lox
var x = "outer";
fun f() { print x; }
f(); // Should print "outer"
x = "changed";
f(); // Should print "changed"
```

Run these tests after each significant change to ensure you haven't introduced regressions.

6. Using a Debugger: Strategic Breakpoints

While print-based logging is helpful, a modern IDE debugger allows you to pause execution, inspect variables, and step through code. Set breakpoints at critical junctions:

- In `Parser.parse()` when a specific token type is consumed.
- In `Interpreter.visitBinaryExpr` when the operator is `PLUS` .
- In `LoxFunction.call()` when a new environment is created.
- In `Environment.get()` when a particular variable name is looked up.

Watch the call stack to understand the flow of control, especially for recursive functions and nested evaluations.

Implementation Guidance

This section provides concrete code snippets to implement the debugging techniques described above. These are supplemental tools that you can integrate into your interpreter during development.

A. Enhanced AST Printer

Add this class to your project. It implements the `Expr.Visitor<String>` and `Stmt.Visitor<String>` interfaces to produce S-expression representations.

JAVA

```java
// In file: tool/AstPrinter.java

package com.craftinginterpreters.lox.tool;

import com.craftinginterpreters.lox.Expr;

import com.craftinginterpreters.lox.Stmt;

import java.util.List;

public class AstPrinter implements Expr.Visitor<String>, Stmt.Visitor<String> {

    public String print(Expr expr) {

        return expr.accept(this);

    }

    public String print(Stmt stmt) {

        return stmt.accept(this);

    }

    @Override

    public String visitBinaryExpr(Expr.Binary expr) {

        return parenthesize(expr.operator.lexeme, expr.left, expr.right);

    }

    @Override

    public String visitGroupingExpr(Expr.Grouping expr) {

        return parenthesize("group", expr.expression);

    }

    @Override

    public String visitLiteralExpr(Expr.Literal expr) {

        if (expr.value == null) return "nil";

        return expr.value.toString();

    }

    @Override

    public String visitUnaryExpr(Expr.Unary expr) {

        return parenthesize(expr.operator.lexeme, expr.right);

    }

    // TODO: Add visit methods for other expression types added later:

    // Variable, Assign, Logical, Call, Get, Set, This, Super
```

```java
    // For example:
    // @Override
    // public String visitVariableExpr(Expr.Variable expr) {
    //     return expr.name.lexeme;
    // }

    @Override
    public String visitExpressionStmt(Stmt.Expression stmt) {
        return parenthesize("expr", stmt.expression);
    }

    @Override
    public String visitPrintStmt(Stmt.Print stmt) {
        return parenthesize("print", stmt.expression);
    }

    @Override
    public String visitVarStmt(Stmt.Var stmt) {
        if (stmt.initializer == null) {
            return parenthesize("var", new Expr.Variable(stmt.name));
        }
        return parenthesize("var", new Expr.Variable(stmt.name), stmt.initializer);
    }

    // TODO: Add visit methods for other statement types added later:
    // Block, If, While, Function, Return, Class

    private String parenthesize(String name, Expr... exprs) {
        StringBuilder builder = new StringBuilder();
        builder.append("(").append(name);
        for (Expr expr : exprs) {
            builder.append(" ");
            builder.append(expr.accept(this));
        }
        builder.append(")");
        return builder;
    }
```

```
    // Helper to print a list of statements
    public String print(List<Stmt> statements) {
        StringBuilder builder = new StringBuilder();
        for (Stmt stmt : statements) {
            builder.append(print(stmt));
            builder.append("\n");
        }
        return builder.toString();
    }
}
```

B. Environment Dumper

Add a static method to the `Environment` class to dump its contents. This method prints the current scope and recursively dumps enclosing scopes with indentation.

JAVA

```
// In file: environment/Environment.java (add this method)
public class Environment {
    // ... existing fields and methods ...

    public void dump() {
        dump(0);
    }

    private void dump(int depth) {
        String indent = "  ".repeat(depth);
        System.out.println(indent + "Environment " + System.identityHashCode(this) + ":");
        for (Map.Entry<String, Object> entry : values.entrySet()) {
            System.out.println(indent + "  " + entry.getKey() + " = " + entry.getValue());
        }
        if (enclosing != null) {
            enclosing.dump(depth + 1);
        } else {
            System.out.println(indent + "  (no enclosing)");
        }
    }
}
```

To use it, call `environment.dump();` at any point in the interpreter, e.g., inside `visitVarStmt` after defining a variable, or inside `LoxFunction.call()` before executing the body.

C. Execution Logging with a Flag

Add a static boolean flag to your `Interpreter` class to enable/disable verbose logging. Then, add logging statements at the beginning of each `visit` method.

```java
// In file: interpreter/Interpreter.java

public class Interpreter implements Expr.Visitor<Object>, Stmt.Visitor<Void> {

    private static final boolean DEBUG = true; // Set to false to disable logs

    private int indentLevel = 0;


    private void log(String message) {

        if (!DEBUG) return;

        String indent = "  ".repeat(indentLevel);

        System.err.println(indent + message);

    }


    @Override

    public Object visitBinaryExpr(Expr.Binary expr) {

        log("visitBinaryExpr: " + expr.operator.lexeme);

        indentLevel++;

        Object left = evaluate(expr.left);

        Object right = evaluate(expr.right);

        indentLevel--;

        log("  left=" + left + ", right=" + right);

        // ... evaluation logic ...

        Object result = ...;

        log("  result=" + result);

        return result;

    }


    // Similarly, add log() calls in other visit methods.

    // Increase indentLevel when entering a node, decrease when leaving.

}
```

For statement execution, you might want to log the statement type. For function calls, log the arguments and the return value.

D. Token Stream Visualizer

A simple utility method in your main `Lox` class can print tokens.

```java
// In file: Lox.java (add this method)

private static void printTokens(List<Token> tokens) {

    for (Token token : tokens) {

        System.out.println(token);

    }

}
```

Call it after scanning and before parsing when you need to debug the scanner.

E. Language-Specific Hints (Java)

- **Identity Hash Code:** Use `System.identityHashCode(object)` to print a unique identifier for an object instance, helpful for distinguishing between different environment objects.
- **Exception Stack Trace:** When catching exceptions (like `RuntimeError`), print the stack trace using `e.printStackTrace()` to see the call path.
- **Java Debugger (JDB):** Learn to use JDB or the debugger integrated into your IDE (IntelliJ IDEA, Eclipse, VS Code). Set breakpoints and inspect variables.

F. Milestone Checkpoint for Debugging

After implementing the core interpreter (Milestone 5), you should be able to run a test program and use the debugging tools to verify internal state.

Test Command: Create a file `test.lox`:

```lox
var a = 1;
{
    var a = 2;
    print a;
}
print a;
```

Run your interpreter on this file.

Expected Output:

```
2
1
```

Debugging Verification:

1. Enable the AST printer and verify the AST structure includes a block statement with a nested variable declaration.
2. Enable environment dumping inside the block. You should see two environments: the inner one with `a = 2` and the outer one with `a = 1`.
3. If the output is reversed (1 then 2), your environment chain might be reversed, or variable resolution might be looking only at the global scope.

G. Debugging Tips Table (Recap of Common Issues)

Symptom	Likely Cause	How to Diagnose	Fix
Infinite recursion in parser	Left-recursive grammar	Print parsing method calls	Ensure base case is `primary()`
Variables not found	Environment chain not searched	Dump environment chain	Implement recursive `get()` and `assign()`
No short-circuit evaluation	Both operands evaluated eagerly	Log evaluation order	Check left operand first in `visitLogicalExpr`
Functions share state across calls	Single environment reused	Print environment identity in each call	Create new `Environment` per call
`this` not bound in methods	Missing `this` in method environment	Dump environment inside method call	Bind `this` in `LoxFunction.bind()`

By incorporating these debugging techniques and referring to the symptom table, you can systematically isolate and fix issues throughout your interpreter's development journey.

10. Future Extensions

> **Milestone(s):** None (forward-looking enhancements)

This section explores the frontiers beyond the core interpreter—potential enhancements that the current architecture can accommodate with modest modifications. Think of the interpreter you've built as a **foundational engine**: robust and complete, but designed with extension points that invite further exploration. These extensions serve dual purposes: they demonstrate the flexibility of your design, and they provide meaningful projects for deepening your understanding of language implementation.

> **Design Principle:** A well-architected interpreter should be **extensible along three dimensions**: new language features, performance improvements, and developer experience enhancements, without requiring fundamental redesigns of the core pipeline.

Possible Feature Additions

The interpreter's modular design—clean separation between lexical analysis, syntactic parsing, and semantic evaluation—creates natural insertion points for new capabilities. Each feature addition follows a consistent pattern: extend the **grammar** (Scanner and Parser), add corresponding **AST nodes** (Data Model), and implement **semantics** (Interpreter visitor methods). The Visitor pattern proves particularly valuable here, as new node types simply require new visitor methods without disturbing existing evaluation logic.

Adding a Read-Eval-Print Loop (REPL)

Mental Model: The REPL transforms the interpreter from a **batch processor** (reading files) into an **interactive conversation partner**, enabling immediate feedback and exploratory programming—like having a calculator that understands Lox syntax.

Current Architecture Fit: The interpreter already has a `Lox.run(String)` method that processes a source string. The REPL adds:

1. A continuous loop in `Lox.runPrompt()` that reads lines from standard input
2. Special handling for multi-line inputs (like function declarations spanning lines)
3. A persistent top-level environment across inputs (unlike file execution which starts fresh)

Required Changes:

Component	Modification Required	Complexity
`Lox` class	Add `runPrompt()` with readline loop, detect incomplete statements	Low
`Parser`	Synchronize on newline when in REPL mode vs. EOF in file mode	Medium
`Interpreter`	Maintain global environment across REPL inputs, maybe display results automatically	Low
Error Handling	Continue after errors in REPL (don't exit entire session)	Medium

Implementation Considerations: The main challenge is handling **incomplete statements**. When the parser encounters EOF mid-statement (like `fun f(`), it must distinguish between "end of file" (error) and "end of line, more input expected" (REL prompt for continuation). A simple approach: catch `ParseError` and if in REPL mode and error was unexpected EOF, read another line and concatenate.

> **ADR: REPL Continuation Strategy**
>
> - **Context:** REPL must handle multi-line inputs gracefully without requiring perfect syntax on first line
> - **Options Considered:**
>   1. **Prompt-based continuation:** When parser hits unexpected EOF, print `... >` and read more input
>   2. **Bracket counting:** Track open `{`, `(`, `[` and continue until all closed
>   3. **Statement delimiter:** Require explicit continuation character (like `\` at line end)
> - **Decision:** Prompt-based continuation with basic bracket counting
> - **Rationale:** Mimics behavior of mature REPLs (Python, Node.js); bracket counting handles most real multi-line cases (function bodies, blocks)
> - **Consequences:** More complex parser error recovery logic; need to reset parser state on continuation

Standard Library Functions

Mental Model: The standard library provides **pre-built tools** that every Lox program can use without reinvention—like a Swiss Army knife included with the language. These are functions written not in Lox, but in Java (or your implementation language) and exposed as built-in callables.

Architecture Integration: The `Environment` class already has a `globals` field. Standard library functions become `LoxFunction` objects (or a new `LoxNativeFunction` subclass) registered in the global environment during interpreter initialization.

Function Category	Example Functions	Implementation Approach
Mathematical	`sqrt(x)`, `sin(x)`, `random()`	Delegate to Java `Math` class, with type checking
String Operations	`length(s)`, `substring(s, start, end)`, `split(s, delim)`	Operate on Java `String` wrapper, return new `LoxString`
Type Utilities	`typeOf(value)`, `isNumber(value)`, `toString(value)`	Inspect runtime object types, use existing `stringify()`
Collection Basics	`arrayCreate(size)`, `arrayPush(arr, value)`	Require new array data type (see below)

Native Function Interface:

```java
// Example interface (shown in Implementation Guidance)
interface LoxCallable {
    int arity();
    Object call(Interpreter interpreter, List<Object> arguments);
}
```

Implementation Pattern: Each standard library function implements `LoxCallable` with type-checked arguments and Java-side logic. The `Interpreter` initializes `globals` with these functions before any user code runs.

Native Functions and Foreign Function Interface (FFI)

Mental Model: A **bridge to the host world**—allowing Lox programs to call functions written in Java (or C/Rust) and access system capabilities like file I/O, networking, or graphics. This transforms Lox from a toy language into one that can interact with the real world.

Architecture Extension: Native functions share the same `LoxCallable` interface as standard library functions but require careful design for data marshalling between Lox values and host language types.

Concern	Design Decision	Rationale
Type Marshalling	Convert Lox values to Java objects: `LoxNumber` → `Double`, `LoxString` → `String`	Simpler for implementer, leverages host language type system
Error Propagation	Throw `RuntimeError` for invalid arguments or host-side failures	Consistent with Lox's error handling model
Memory Management	Native functions can allocate host resources; need cleanup hooks	Simple approach: no automatic cleanup (educational focus)
Callback Support	Allow native functions to call back into Lox functions	Requires passing `LoxFunction` as argument, maintaining interpreter context

> **ADR: FFI Safety vs. Power Trade-off**
>
> - **Context:** Native functions can crash the interpreter or cause undefined behavior if misused
> - **Options Considered:**
>   1. **Restricted FFI:** Only pre-approved safe functions (math, string ops)
>   2. **Unrestricted FFI:** Allow any Java method call via reflection
>   3. **Sandboxed FFI:** Run native code in isolated environment with resource limits
> - **Decision:** Restricted FFI with explicit registration of safe functions
> - **Rationale:** Educational project prioritizes safety and simplicity; reflection adds complexity without proportional learning value
> - **Consequences:** Limited to functions we explicitly implement; can't dynamically call arbitrary Java methods

Additional Control Flow: `break`, `continue`, and `switch`

Mental Model: `break` and `continue` are **loop control commands**—emergency exits and fast-forward buttons for loops. `switch` is a **multi-way branch**—a more structured alternative to chained `if-else` statements.

Grammar Extensions:

```
statement → exprStmt | printStmt | block | ifStmt | whileStmt | forStmt
          | breakStmt | continueStmt | switchStmt ;

breakStmt → "break" ";" ;
continueStmt → "continue" ";" ;

switchStmt → "switch" "(" expression ")" "{"
             ( "case" expression ":" statement* )*
             ( "default" ":" statement* )?
           "}" ;
```

Implementation Challenges:

Feature	Challenge	Solution Approach
`break` / `continue`	Must exit/continue the innermost loop, not just any statement	Store loop nesting depth in interpreter, use exception-like control flow
`switch`	Multiple cases with fall-through vs. break behavior	Implement Java-style fall-through (educational) or require explicit `break`
Scope	Do cases create new scopes?	Each case block gets its own environment for consistency with `if`

Control Flow Implementation Pattern: Use a specialized exception (`BreakException` , `ContinueException`) that carries no value but unwinds the call stack to the nearest loop boundary. The `visitWhileStmt` and `visitForStmt` methods catch these exceptions and handle them appropriately.

Arrays and Basic Data Structures

Mental Model: Arrays are **indexed collections**—numbered slots that can hold values of any type. Adding arrays transforms Lox from a calculator with variables into a language capable of handling real data.

Architecture Impact:

1. **New Token Types:** `LEFT_BRACKET` , `RIGHT_BRACKET`
2. **New AST Nodes:** `ArrayLiteral` (list of expressions), `SubscriptExpr` (array[index])
3. **New Runtime Type:** `LoxArray` with internal `List<Object>` storage
4. **New Operations:** Index get/set, `length` property/method

Grammar:

```
primary → ( "[" ( expression ( "," expression )* )? "]" ) | ... existing rules ... ;
postfix → primary ( "[" expression "]" | "." IDENTIFIER | "(" arguments? ")" )* ;
```

Design Decisions for Arrays:

Decision Point	Options	Recommended Choice
Index Origin	0-based (C/Java) vs 1-based (Lua)	0-based for consistency with implementation language
Bounds Checking	Check at runtime, throw error if out of bounds	Always check; no buffer overflows
Dynamic Resizing	Fixed-size vs growable	Growable via `add()` method for simplicity
Multi-dimensional	Arrays of arrays vs native multi-D	Arrays of arrays (more flexible)

Integration: The `LoxArray` class implements `LoxInstance` so arrays can have methods (`length()` , `push(value)` , `pop()`) while also supporting indexed access via `array[0]` syntax.

Exception Handling with `try` / `catch`

Mental Model: Exceptions are **controlled crashes**—a way for functions to signal "I can't handle this situation" and let callers decide what to do. The `try` block is a safety net, and `catch` is the person who catches what falls.

Grammar Extension:

```
statement → ... | tryStmt ;
tryStmt → "try" block "catch" "(" IDENTIFIER ")" block ;
```

Implementation Approach:

1. Add `RuntimeError` subclass `LoxException` that carries a value (the exception object)
2. In `Interpreter` , when evaluating an expression that throws, unwind until finding a `try` block
3. The `catch` clause creates a new local variable (the identifier) bound to the exception value

4. Execute the catch block in that environment

Design Considerations:

Consideration	Decision	Rationale
Exception Values	Any Lox value can be thrown	Simple, flexible; follows JavaScript model
Finally Clause	Omit for simplicity	Adds significant complexity for cleanup
Exception Hierarchy	Single exception type vs class hierarchy	Single `LoxException` for educational simplicity
Uncaught Exceptions	Terminate program with error message	Same as current runtime error behavior

Modules and Imports

Mental Model: Modules are **self-contained toolboxes**—collections of related functions and classes that can be loaded on demand. The `import` statement is like saying "I need the tools from that toolbox."

Architecture Changes Required:

Component	Changes	Complexity
Scanner/Parser	New `import` keyword and statement	Low
Interpreter	Module cache, separate environments per module	High
File System	Resolve module paths, detect cycles	Medium
Environment	Module-level vs global vs local scopes	High

Simplified Approach (Single File Modules):

1. `import "module.lox"` loads and executes that file once
2. Creates a new `Environment` for the module (parent is globals)
3. Exports: all top-level declarations become available to importer
4. Import returns a `LoxModule` object that provides access to exported names

Implementation Pattern: Use a `Map<String, Environment>` module cache. When importing, check cache first; if not present, run the module file's statements in a fresh environment, store that environment in cache, and return a wrapper that delegates lookups to it.

Compiling to Bytecode (Transition to a Virtual Machine)

Mental Model: Bytecode compilation transforms the **AST interpreter** (walking the tree each time) into a **virtual machine** (walking a linear instruction list). Think of it as converting a recipe (AST) into a punch card for a cooking robot (bytecode VM)—more steps upfront, but faster execution.

Architecture Transformation Path:

Step	Description	Impact
1. **Define Bytecode Instruction Set**	Simple stack-based operations: `CONSTANT`, `ADD`, `STORE`, `LOAD`	New `OpCode` enum, `Chunk` class
2. **Write Compiler Visitor**	Transform AST nodes to bytecode sequences	New `Compiler` class implementing visitor interfaces
3. **Build Virtual Machine**	Stack machine that executes bytecode	New `VM` class replacing parts of `Interpreter`
4. **Gradual Migration**	Start with expressions only, keep statements as tree-walking	Hybrid approach reduces risk

Bytecode Example: The expression `1 + 2 * 3` compiles to:

```
CONSTANT 0  (value: 1)
CONSTANT 1  (value: 2)
CONSTANT 2  (value: 3)
MULTIPLY
ADD
```

Performance Benefits: Bytecode eliminates AST traversal overhead, enables optimization passes (constant folding, dead code elimination), and prepares for Just-In-Time (JIT) compilation.

> **ADR: Gradual vs Complete VM Migration**
>
> - **Context:** Transitioning from tree-walking to bytecode VM is a major architectural change
> - **Options Considered:**
>   1. **Complete Rewrite:** Build VM from scratch, discard tree-walking interpreter
>   2. **Gradual Migration:** Compile expressions to bytecode, interpret statements as AST
>   3. **Dual Implementation:** Support both interpreters, select via flag
> - **Decision:** Gradual migration starting with expressions
> - **Rationale:** Lower risk, allows testing VM incrementally, preserves educational value of seeing both approaches
> - **Consequences:** Temporary complexity of two evaluation strategies; need bridge between VM and environment systems

Performance Optimizations

While the tree-walking interpreter prioritizes clarity over speed, several optimizations can significantly improve performance without altering the fundamental architecture. These optimizations demonstrate important compiler techniques while keeping the codebase comprehensible.

Constant Folding

Mental Model: Constant folding is the interpreter's **pre-calculation**—evaluating constant expressions at compile time rather than runtime, like a baker measuring all ingredients before starting rather than stopping to measure between each step.

Implementation Approach: Add a compile-time optimization pass between parsing and interpretation that walks the AST, identifying sub-expressions with only literal values, evaluating them, and replacing with a single `Literal` node.

Example Transformation:

- **Before:** `Binary(+, Literal(1), Binary(*, Literal(2), Literal(3)))`
- **After:** `Literal(7)`

Algorithm:

1. Create `ConstantFolder` class implementing `Expr.Visitor<Expr>` and `Stmt.Visitor<Stmt>`
2. For each `Binary` node: recursively fold children; if both are literals, compute result, return new `Literal`
3. Apply to `Unary`, `Grouping`, and `Logical` nodes similarly
4. For statements: fold expressions within `Print`, `Var`, `If`, `While`, etc.

Limitations: Cannot fold variables (unknown at compile time) or function calls (side effects). Safe to apply only to pure arithmetic and logical operations.

Environment Flattening (Upvalue Resolution)

Mental Model: Environment flattening transforms the **linked list of scopes** (slow chain of hash map lookups) into a **flat array of slots** (fast indexed access). Think of it as creating a cheat sheet that lists all variables you'll need during a function's execution, with their "home addresses" pre-calculated.

Current Performance Issue: Each variable lookup in a closure may traverse multiple environment links: local → closure-captured → outer function → global. With nested closures, this becomes O(depth) per access.

Optimization Approach:

1. **Analysis Phase:** Walk function body AST before execution, identify all variable references

2. **Upvalue Resolution:** For variables not defined locally, record which enclosing environment and depth they come from
3. **Flat Storage:** Store all accessed variables (local and upvalues) in a single array
4. **Fast Access:** Variable references become array indices instead of name lookups

Implementation Steps:

1. Add `Resolver` pass before interpretation (similar to book's Chapter 11 resolver)
2. Store resolution results in `Interpreter.locals: Map<Expr, Integer>` (already in design!)
3. Modify `Environment` to support indexed access alongside name-based access
4. Change `visitVariableExpr` to use index when available

Performance Impact: Variable access changes from O(n) hash lookup + chain traversal to O(1) array access. Most significant for loops accessing captured variables.

Method Caching

Mental Model: Method caching is the interpreter's **Rolodex**—remembering where you found a method last time so you don't have to search the class hierarchy again. For `object.method()` calls, cache the resolved method at the call site.

Problem: Each `object.method()` call traverses: instance fields → class methods → superclass chain. With inheritance hierarchies, this repeated traversal adds overhead.

Solution: Add a **cache at the call site** (AST node) that remembers: for a given receiver object's class, which method was found. On subsequent calls with same class, use cached method.

Implementation:

1. Add field to `Expr.Call` node: `cachedMethod: LoxFunction` and `cachedClass: LoxClass`
2. In `visitCallExpr`, before full resolution: check if `callee` is `Get` expression and if receiver's class matches cached class
3. If match, use cached method; if not, perform full resolution and update cache

Challenge: Cache invalidation when classes are redefined (unlikely in Lox) or when monkey-patching methods (not supported). Simple approach: clear all caches on any class declaration (heavy-handed but correct).

String Interning

Mental Model: String interning is the interpreter's **dictionary**—ensuring identical strings share the same underlying object, so comparisons become pointer equality checks instead of character-by-character comparisons.

Performance Benefits:

1. **Faster equality checks:** `==` on strings becomes reference comparison for interned strings
2. **Reduced memory:** Duplicate string literals share storage
3. **Faster hash map lookups:** Environment variable names are interned strings

Implementation: Add `StringPool` class with static `intern(String): String` method. In `Scanner`, after reading string literal text, pass through pool before creating `Token`. In `Interpreter`, intern string values created at runtime (concatenation results).

Trade-off: Interning pool grows indefinitely (memory leak for dynamic strings). Solution: weak references or limit to literals only.

Tail Call Optimization (TCO)

Mental Model: Tail call optimization transforms **recursive function calls** that would grow the call stack into **loops** that reuse the current stack frame. It's like realizing you're at the end of a hallway and instead of walking back to start a new hallway, you just turn around and continue.

Applicability: When a function's last action is returning the result of another function call (tail position), reuse the current call frame instead of pushing a new one.

Example:

```lox
fun factorial(n, acc) {
    if (n <= 1) return acc;
    return factorial(n - 1, n * acc);  // Tail call eligible for TCO
}
```

Implementation: In `LoxFunction.call()`, detect tail calls: if the last statement in function body is `return expr` and `expr` is a `Call` expression, then:

1. Don't create new environment; reuse current with updated parameters
2. Jump to evaluating the called function's body (loop instead of recursion)
3. Implement as trampoline: functions return "continue with this other function" instead of recursing

Benefit: Enables infinite recursion without stack overflow for properly tail-recursive functions.

Arithmetic Operation Specialization

Mental Model: Operation specialization creates **fast paths** for common cases—like having a special "add integers" circuit in a calculator instead of using the general "add floating point" logic every time.

Current Implementation: All numbers are `Double`, all arithmetic uses Java's double operators. This is simple but inefficient for integer-heavy code.

Optimizations:

1. **Integer Detection:** If both operands are integers (no decimal part), use integer arithmetic
2. **Type Tagging:** Store numbers as `long` with type tag in low bits (NaN-boxing)
3. **Specialized Ops:** `ADD_INT`, `SUBTRACT_INT` bytecode instructions (if moving to VM)

Simple Approach: In `visitBinaryExpr`, check if both operands are integers (use `Math.floor()` comparison), perform integer arithmetic, convert back to double. Avoids floating point rounding for integer cases.

Implementation Guidance

Technology Recommendations

Extension	Simple Implementation	Advanced Implementation
REPL	`java.util.Scanner` for line input	JLine3 for history, completion, editing
Standard Library	Static methods in `LoxStdLib` class	Dynamic loading via service provider interface
Native Functions	Hardcoded registry of `LoxCallable`	Reflection-based discovery with annotations
Arrays	`LoxArray` with `ArrayList<Object>` backend	Custom resizable array with type tagging
Bytecode VM	Stack VM with 30-40 opcodes	Register VM with optimization passes

Recommended File Structure for Extensions

```
lox/
├── interpreter/          # Core interpreter (existing)
│   ├── Interpreter.java
│   ├── Environment.java
│   └── ...
├── extensions/           # New directory for optional extensions
│   ├── repl/
│   │   ├── Repl.java              # Enhanced REPL with continuation
│   │   └── LineBuffer.java        # Multi-line input buffer
│   ├── stdlib/
│   │   ├── LoxStdLib.java         # Standard function registry
│   │   ├── MathFunctions.java     # sqrt, sin, random
│   │   ├── StringFunctions.java   # length, substring
│   │   └── TypeFunctions.java     # typeOf, toString
│   ├── native/
│   │   ├── NativeRegistry.java    # Maps names to native functions
│   │   ├── FileNative.java        # File I/O functions
│   │   └── TimeNative.java        # Time/date functions
│   ├── arrays/
│   │   ├── LoxArray.java          # Array runtime type
│   │   └── ArrayCompiler.java     # Compiler for [ ] syntax
│   └── vm/                         # Bytecode VM extension
│       ├── OpCode.java
│       ├── Chunk.java
│       ├── VM.java
│       └── Compiler.java
└── Lox.java              # Modified to load extensions
```

Infrastructure Starter Code: Native Function Interface

```java
// extensions/native/NativeCallable.java
package lox.extensions.native;

import lox.Interpreter;
import java.util.List;

/**
 * Interface for native (Java-implemented) functions callable from Lox.
 * All standard library and extension functions implement this.
 */
public interface NativeCallable {
    /**
     * Number of arguments this function expects.
     */
    int arity();

    /**
     * Call the function with the given arguments.
     *
     * @param interpreter The current interpreter context
     * @param arguments The evaluated argument values
     * @return The function's result
     */
    Object call(Interpreter interpreter, List<Object> arguments);

    /**
     * Human-readable name for debugging.
     */
    default String name() {
        return getClass().getSimpleName();
    }
}

// extensions/native/NativeRegistry.java
package lox.extensions.native;
```

```java
import lox.Environment;
import java.util.HashMap;
import java.util.Map;

/**
 * Registry of all available native functions.
 * Populates the global environment with these functions.
 */
public class NativeRegistry {
    private final Map<String, NativeCallable> functions = new HashMap<>();

    public NativeRegistry() {
        // Register core standard library
        register("clock", new ClockFunction());
        register("sqrt", new MathFunction("sqrt", Math::sqrt));
        // ... more registrations
    }

    public void register(String name, NativeCallable function) {
        functions.put(name, function);
    }

    public void installInto(Environment globals) {
        // Wrap each NativeCallable in a LoxCallable adapter
        for (Map.Entry<String, NativeCallable> entry : functions.entrySet()) {
            globals.define(entry.getKey(), new NativeAdapter(entry.getValue()));
        }
    }

    /**
     * Adapter that makes NativeCallable work with existing LoxCallable interface.
     */
    private static class NativeAdapter implements lox.LoxCallable {
        private final NativeCallable nativeFn;

        NativeAdapter(NativeCallable nativeFn) {
```

```
            this.nativeFn = nativeFn;
        }

        @Override
        public int arity() {
            return nativeFn.arity();
        }

        @Override
        public Object call(Interpreter interpreter, List<Object> arguments) {
            return nativeFn.call(interpreter, arguments);
        }

        @Override
        public String toString() {
            return "<native fn " + nativeFn.name() + ">";
        }
    }
}

// Example native function: clock() returns seconds since epoch
class ClockFunction implements NativeCallable {
    @Override
    public int arity() {
        return 0;  // clock() takes no arguments
    }

    @Override
    public Object call(Interpreter interpreter, List<Object> arguments) {
        // Return as Lox number (double)
        return (double) System.currentTimeMillis() / 1000.0;
    }
}
```

Core Logic Skeleton: Constant Folding Optimizer

```java
// extensions/optimization/ConstantFolder.java

package lox.extensions.optimization;

import lox.*;
import lox.Stmt;
import java.util.List;

/**
 * AST optimizer that performs constant folding.
 * Replaces constant expressions with their computed values.
 */
public class ConstantFolder implements Expr.Visitor<Expr>, Stmt.Visitor<Stmt> {

    // TODO 1: Implement visitBinaryExpr to fold constant binary operations
    //   - Recursively fold left and right operands first
    //   - Check if both are Literal nodes after folding
    //   - If yes, compute result based on operator:
    //       PLUS: add values (handle string concatenation)
    //       MINUS, STAR, SLASH: arithmetic (check division by zero)
    //       GREATER, LESS, etc.: comparisons
    //   - Return new Literal with computed value, or original if not foldable

    @Override
    public Expr visitBinaryExpr(Expr.Binary expr) {
        // TODO 1.1: Fold left operand
        Expr left = expr.left.accept(this);
        // TODO 1.2: Fold right operand
        Expr right = expr.right.accept(this);

        // TODO 1.3: Check if both are Literal after folding
        if (left instanceof Expr.Literal && right instanceof Expr.Literal) {
            Object leftVal = ((Expr.Literal) left).value;
            Object rightVal = ((Expr.Literal) right).value;

            // TODO 1.4: Based on operator type, compute result
```

```
        switch (expr.operator.type) {
            case PLUS:
                // TODO: Handle number addition and string concatenation
                break;
            case MINUS:
                // TODO: Check both are numbers, subtract
                break;
            // ... handle other operators
        }
    }

    // TODO 1.5: If folded, return new Literal, otherwise new Binary with folded children
    return null; // placeholder
}

// TODO 2: Implement visitUnaryExpr for - and ! on literals

// TODO 3: Implement visitGroupingExpr to fold grouped expressions

// TODO 4: Implement visitLogicalExpr with short-circuit awareness
//   - Fold left first
//   - If left is literal and operator is OR with truthy left, return left
//   - If left is literal and operator is AND with falsy left, return left
//   - Otherwise fold right and rebuild

// TODO 5: Implement statement visitors to fold expressions within statements
//   visitPrintStmt, visitVarStmt, visitIfStmt, etc.

// TODO 6: Add main fold() method that processes a list of statements
public List<Stmt> fold(List<Stmt> statements) {
    // TODO: Apply visitor to each statement, collect results
    return null; // placeholder
}

// Helper method to check if value is numeric
```

```java
    private boolean isNumber(Object value) {
        return value instanceof Double;
    }

    // Helper method to check if expression is pure (no side effects)
    // Used to decide if folding is safe
    private boolean isPure(Expr expr) {
        // TODO: Walk expression tree, return false if contains:
        //   - Variable expressions (values unknown at compile time)
        //   - Function calls (may have side effects)
        //   - Assignments (side effects)
        return false; // placeholder
    }
}
```

Core Logic Skeleton: REPL with Continuation

```java
// extensions/repl/Repl.java

package lox.extensions.repl;


import lox.*;

import java.util.Scanner;

import java.util.ArrayList;

import java.util.List;


/**

 * Enhanced REPL with multi-line input support.

 */

public class Repl {

    private final Interpreter interpreter;

    private final Scanner scanner;

    private boolean hadError = false;

    private boolean inMultilineMode = false;

    private final StringBuilder multilineBuffer = new StringBuilder();



    public Repl(Interpreter interpreter) {

        this.interpreter = interpreter;

        this.scanner = new Scanner(System.in);

    }



    public void run() {

        System.out.println("Lox REPL (type 'exit' to quit)");



        while (true) {

            // TODO 1: Display appropriate prompt

            String prompt = inMultilineMode ? "... > " : "> ";

            System.out.print(prompt);



            // TODO 2: Read line, handle EOF (Ctrl+D)

            if (!scanner.hasNextLine()) {

                System.out.println();

                break;
```

```
        }
        String line = scanner.nextLine().trim();


        // TODO 3: Check for exit command
        if (line.equals("exit") || line.equals("quit")) {
            break;
        }


        // TODO 4: Handle empty line in multiline mode (ends input)
        if (inMultilineMode && line.isEmpty()) {
            processBuffer();
            continue;
        }


        // TODO 5: Append to buffer (with newline if in multiline mode)
        if (inMultilineMode) {
            multilineBuffer.append(line).append("\n");
        } else {
            multilineBuffer.append(line);
        }


        // TODO 6: Try to parse current buffer
        try {
            run(multilineBuffer.toString());
            // Success: reset state
            multilineBuffer.setLength(0);
            inMultilineMode = false;
        } catch (ParseError error) {
            // TODO 7: Check if error is due to incomplete input
            //   - If at EOF and expecting more tokens (like unclosed brace)
            //   - Set inMultilineMode = true and continue
            //   - Otherwise, report error and reset
            if (isIncompleteError(error)) {
                inMultilineMode = true;
            } else {
```

```
                // Report error
                multilineBuffer.setLength(0);
                inMultilineMode = false;
            }
        }
    }

    scanner.close();
}

private void run(String source) {
    // TODO: Use existing Lox.run() logic but don't exit on error
    //   - Create scanner, parser
    //   - Parse statements
    //   - If parse error, throw
    //   - Execute with interpreter
    //   - Catch runtime errors, print but don't exit
}

private boolean isIncompleteError(ParseError error) {
    // TODO: Heuristic: error message contains "expect" and token is EOF
    //   or unclosed delimiter {, (, [
    return false; // placeholder
}

private void processBuffer() {
    // Force parse attempt even with empty line continuation
    try {
        run(multilineBuffer.toString());
    } catch (ParseError e) {
        // Give up on this input
        System.out.println("Syntax error: " + e.getMessage());
    } finally {
        multilineBuffer.setLength(0);
        inMultilineMode = false;
```

```
        }
    }
}
```

Language-Specific Hints (Java)

1. **REPL Input:** Use `java.util.Scanner` for simplicity, but it doesn't support arrow keys or history. For better UX, consider `jline3` dependency:

```java
LineReader reader = LineReaderBuilder.builder().build();
String line = reader.readLine("> ");
```

2. **Native-Java Integration:** For exposing Java objects to Lox, use the `invoke` method from `java.lang.reflect` but beware of security and complexity. Better to create explicit wrapper classes for safety.
3. **Performance Measurement:** Use `System.nanoTime()` before/after execution to benchmark optimizations:

```java
long start = System.nanoTime();
interpreter.interpret(statements);
long elapsed = System.nanoTime() - start;
System.out.printf("Executed in %.2f ms%n", elapsed / 1_000_000.0);
```

4. **Memory Profiling:** For array/string interning, consider using `java.lang.ref.SoftReference` for cache entries that can be GC'd under memory pressure.

Milestone Checkpoints for Extensions

Extension	Verification Command	Expected Output
REPL	Run `java Lox` (no args)	Shows `>` prompt, executes `print 1+2;` prints `3`
Standard Library	`print clock();`	Prints current timestamp (number)
Arrays	`var a = [1, 2, 3]; print a[1];`	Prints `2`
Constant Folding	Test with `print 2 * 3 + 4 * 5;`	Prints `26`, check AST shows single literal
Native Functions	`fun readFile(path) { ... }` (native)	Can read and print file contents

Debugging Tips for Extensions

Symptom	Likely Cause	How to Diagnose	Fix
REPL exits after one line	Scanner reading full input as one line	Add debug print of raw input string	Use `Scanner.nextLine()` correctly
Native function returns wrong type	Marshalling between Java/Lox types incorrect	Print Java value before returning	Ensure numbers as Double, strings as LoxString
Constant folding changes program behavior	Folded expression with side effects	Check `isPure()` heuristic	Don't fold function calls or variable accesses
Array index out of bounds	No bounds checking in `LoxArray.get()`	Add check before array access	Throw `RuntimeError` with index and length
Memory leak with string interning	Interning pool never clears entries	Monitor heap usage with VisualVM	Use `WeakHashMap` or limit to literals
Tail recursion still overflows stack	TCO not detecting all tail calls	Add debug log of call frame creation	Check `return func(args)` in tail position
Method cache returns wrong method	Cache not invalidated on class redefinition	Print cache hits/misses	Clear cache in `visitClassStmt`

---

11. Glossary

> **Milestone(s):** All milestones (reference section)

This glossary provides definitions for the key technical terms, acronyms, and domain-specific vocabulary used throughout this design document. Terms are organized alphabetically for easy reference.

A

Abstract Syntax Tree (AST) A hierarchical tree representation of the grammatical structure of source code, where each node corresponds to a language construct (expression, statement) and child nodes represent its components. The AST discards surface syntax details like parentheses and whitespace, focusing on the essential syntactic relationships.

Arity The number of parameters a function or callable expects. In Lox, functions have a fixed arity determined by their parameter list declaration. The `LoxFunction.arity()` method returns this value, enabling the interpreter to check argument counts before calling.

Array An indexed collection data type that stores elements in a contiguous, numerically-indexed sequence. While not in the base Lox language described in "Crafting Interpreters," it's a common extension where arrays support operations like getting, setting, and checking length.

AST Node Classes The concrete Java classes that implement the `Expr` and `Stmt` interfaces, representing specific language constructs in the AST. Examples include `Binary` (for binary operations), `Unary` (for unary operations), `Literal` (for literal values), `If` (for conditional statements), and `Function` (for function declarations).

AST Printer A visitor implementation that converts an AST to a human-readable string representation, typically using parenthesized S-expression notation. Useful for debugging the parser by visualizing the structure it produces.

B

Binary Expression An expression with two operands and a single operator between them, such as `1 + 2` or `x < y`. Represented by the `Binary` AST node with `left`, `operator`, and `right` fields.

Block A sequence of statements enclosed in curly braces `{}` that creates a new lexical scope. Represented by the `Block` AST node containing a list of statements. When executed, it creates a new nested `Environment`.

Boolean A primitive truth value in Lox, either `true` or `false`. Represented at runtime as a `LoxBoolean` object wrapping a Java `Boolean`. Subject to Lox's truthiness rules in conditional contexts.

Break Statement A control flow statement that immediately terminates execution of the innermost enclosing loop. While not in the base Lox language, it's a common extension implemented via a `BreakException` that unwinds the call stack until caught by the loop's execution machinery.

Bytecode A compact, intermediate representation of a program designed for efficient execution by a virtual machine. Each instruction (opcode) typically performs a simple operation like push constant, add, or jump. Contrasts with tree-walking interpretation, which directly traverses the AST.

C

Call The act of invoking a function, method, or class constructor with arguments. Represented by the `Call` AST node with `callee` (the function expression), `paren` (the closing parenthesis token for error reporting), and `arguments` (list of argument expressions).

Chunk In bytecode interpreter architectures, a container for a sequence of bytecode instructions and their associated constant pool. The `Chunk` class stores the instruction stream and constant values referenced by those instructions.

Class A blueprint for creating objects (instances) that encapsulates data (fields) and behavior (methods). In Lox, a class declaration creates a `LoxClass` object that can be called as a constructor to create `LoxInstance` objects. Classes support single inheritance via the `<` syntax.

Closure A first-class function value that "closes over" (captures) variables from its lexical (surrounding) scope, maintaining access to those variables even after the enclosing function has returned. Implemented by storing a reference to the function's defining `Environment` within the `LoxFunction` object.

Comparison Operators Binary operators that compare two values and return a boolean: `<` (less than), `>` (greater than), `<=` (less than or equal), `>=` (greater than or equal). These have higher precedence than equality operators but lower than addition/subtraction.

Constant Folding A compiler optimization that evaluates constant expressions at compile time (parse time) rather than runtime. For example, the expression `3 + 4 * 2` can be computed once as `11` and replaced with a literal `11` in the AST, reducing runtime work.

Continue Statement A control flow statement that immediately skips to the next iteration of the innermost enclosing loop. While not in the base Lox language, it's a common extension implemented via a `ContinueException` caught by the loop execution.

D

Deep Equality An equality comparison that compares the semantic value of objects rather than their object identity (reference equality). In Lox, `isEqual()` handles special cases: numbers are compared with a tolerance for floating-point errors, strings by character content, and `nil` equals only itself.

Defining Environment The `Environment` that was active when a function was declared, captured by closures to provide lexical scoping. When a closure is called, this environment becomes the parent of the call's new environment, allowing access to outer variables.

Desugaring The process of transforming higher-level syntactic constructs into lower-level primitive constructs. In Lox, `for` loops are desugared into equivalent `while` loop constructs with initialization and increment statements, simplifying the interpreter's implementation.

Division by Zero A runtime error that occurs when the right operand of a division (`/`) operator evaluates to zero. The interpreter detects this and raises a `RuntimeError` with an appropriate message, halting execution.

Double Dispatch A technique where an operation's behavior is determined by both the type of the receiver and the type of the argument. The Visitor pattern uses double dispatch: the AST node's `accept()` method calls the appropriate `visitXxx()` method on the visitor, selecting based on both the visitor type and the node type.

Dynamic Typing A type system where values carry their type information at runtime, and type checking occurs during program execution rather than compile time. Lox is dynamically typed: variables have no declared type, and operations check operand types at evaluation time.

E

End-of-file (EOF) A special token type (`EOF`) that marks the end of the source code input. The scanner adds this token after processing all characters, and the parser uses it to know when to stop parsing.

Environment A runtime data structure that maps variable names to their current values within a specific lexical scope. Implemented as the `Environment` class with a `values` hash map and an optional reference to an `enclosing` parent environment, forming a chain for variable resolution.

Environment Chain The linked list of `Environment` objects representing nested lexical scopes, from the current local scope outward through enclosing function scopes to the global scope. Variable lookup proceeds up this chain until finding the name or reaching the global scope without it (causing a runtime error).

Environment Dumper A debugging utility that prints all variable bindings in the environment chain, typically with indentation to show nesting. Helps diagnose scoping issues by revealing which variables are defined in each scope and their current values.

Environment Flattening An optimization technique that converts the chain of linked `Environment` objects into a flat array indexed by scope depth and variable slot, reducing variable lookup from O(n) in chain length to O(1) array access. Used in more advanced interpreters and compilers.

Equality Operators Binary operators that test for equality (`==`) or inequality (`!=`) between two values. These have the lowest precedence among comparison operators and implement Lox's deep equality semantics.

Exception A special value or object that can be "thrown" to signal an error or exceptional condition and "caught" by exception handling code. While not in base Lox, extensions might add `try` / `catch` statements with `Exception` objects carrying error information.

Execution Log A debug trace of the interpreter's step-by-step evaluation, showing which AST nodes are visited, what values they produce, and how the environment changes. Implemented by adding logging statements to visitor methods, often controlled by a `DEBUG` flag.

Expression A piece of syntax that produces a value when evaluated. Examples include literals (`5` , `"hello"`), variable references (`x`), arithmetic operations (`a + b`), and function calls (`f(3)`). Represented by the `Expr` abstract class and its concrete subclasses.

Expression Statement A statement that consists solely of an expression followed by a semicolon. The expression is evaluated for its side effects (like assignment or function calls), and the resulting value is discarded. Represented by the `Expression` AST node.

F

Fail-Fast An error handling strategy where the interpreter halts execution immediately upon detecting a semantic violation (type mismatch, undefined variable, etc.), rather than attempting to continue with potentially corrupted state. Lox uses fail-fast for runtime errors.

Field A named property stored on a class instance (`LoxInstance`), accessible via dot notation (`instance.field`). Fields are dynamically created when assigned to and stored in the instance's `fields` hash map, separate from methods defined on the class.

First-Class Function A function that can be treated like any other value: assigned to variables, passed as arguments to other functions, returned from functions, and stored in data structures. Lox functions are first-class, implemented as `LoxFunction` objects.

For Loop A control flow statement with initialization, condition, and increment expressions: `for (var i = 0; i < 10; i = i + 1) { ... }` . In Lox, this is desugared into a block containing the initializer, a `while` loop with the condition, and the increment as the last statement in the loop body.

Foreign Function Interface (FFI) A mechanism for calling functions implemented in the host language (Java) from Lox code. Typically implemented via a `NativeCallable` interface and a `NativeRegistry` that maps names to native implementations, which can then be installed into the global environment.

Function A reusable block of code that takes parameters, performs computations, and optionally returns a value. In Lox, functions are declared with the `fun` keyword, creating a `LoxFunction` object that captures its defining environment (for closures). Functions are called with arguments bound to parameters in a new environment.

Function Call The runtime operation of invoking a function with concrete argument values. The interpreter evaluates the argument expressions, creates a new `Environment` with the function's closure environment as parent, binds parameters to argument values, and executes the function body.

G

Grammar Rules The formal specification that defines which sequences of tokens constitute syntactically valid programs in the language. Expressed in a notation like Backus-Naur Form (BNF) or Parsing Expression Grammar (PEG), these rules guide the recursive descent parser's implementation.

Grouping Expression An expression wrapped in parentheses `(expr)`, used to explicitly control operator precedence. Represented by the `Grouping` AST node with a single `expression` child. The parser handles grouping by matching `(` tokens and recursively parsing the inner expression.

H

High-Level Architecture The macro-organization of the interpreter system into major components (Scanner, Parser, Interpreter) and their data flow relationships, as described in Section 3. This architecture defines the pipeline from source text to execution output.

I

Identifier A token representing a name defined by the programmer, such as a variable, function, or class name. The `IDENTIFIER` token type encompasses all names that aren't language keywords. Identifiers follow specific lexical rules (starting with letter/underscore, containing letters/numbers/underscores).

If/Else Statement A conditional control flow statement that executes one of two branches based on a condition's truthiness: `if (condition) thenBranch else elseBranch`. The `else` clause is optional. Represented by the `If` AST node with `condition`, `thenBranch`, and optional `elseBranch` fields.

Import Statement A statement that loads and executes code from another module, typically making its exported definitions available in the current scope. While not in base Lox, extensions might add `import` statements that load Lox source files and return a module object.

Inheritance The mechanism by which a class (the subclass or derived class) acquires properties and methods from another class (the superclass or base class). Lox supports single inheritance via the `<` syntax in class declarations. Method resolution proceeds up the inheritance chain.

Initializer A special method named `init` defined inside a class that is automatically called when an instance is created (via the class constructor call). It receives the constructor arguments and can perform setup, including calling `super.init()` for inheritance chains.

Instance A runtime object created from a class blueprint, with its own set of fields (stored in `LoxInstance.fields`) and access to methods defined on its class (via `LoxClass.methods`). Created by calling the class as a constructor function.

Interpreter The core execution engine that walks the AST and performs the operations it represents. In this design, the `Interpreter` class implements the `Expr.Visitor<Object>` and `Stmt.Visitor<Void>` interfaces, providing `visitXxx()` methods that define the semantics for each AST node type.

J

JIT Compilation (Just-In-Time Compilation) An advanced optimization technique where bytecode or AST is compiled to native machine code at runtime, just before execution, potentially yielding significant performance gains. Contrasts with ahead-of-time compilation and pure interpretation.

K

Keyword A reserved word in the language with special syntactic meaning, such as `var`, `fun`, `if`, `while`, `class`, etc. The scanner recognizes keywords and emits corresponding token types (e.g., `VAR`, `FUN`, `IF`) rather than `IDENTIFIER` tokens.

L

Left-Recursive Grammar A grammar rule where the leftmost symbol in a production is the nonterminal being defined, e.g., `expression → expression + term`. Naive recursive descent parsers get stuck in infinite recursion with left-recursive rules, requiring transformation to right-recursive or iterative forms.

Lexeme The raw character sequence (text) of a token as it appears in the source code. For example, in the token for the number `123.45`, the lexeme is the string `"123.45"`. Stored in the `Token.lexeme` field for debugging and error reporting.

Lexical Analysis The process of converting a sequence of characters (source code) into a sequence of meaningful tokens, performed by the scanner/lexer. This includes recognizing identifiers, keywords, literals, and operators while ignoring whitespace and comments.

Lexical Scoping A scoping discipline where variable visibility is determined by the textual (lexical) structure of the source code: a variable is visible within the block where it's defined and any nested blocks, but not in enclosing or sibling blocks. Lox uses lexical scoping for variables and

closures.

Lexer Synonym for Scanner. See **Scanner**.

Literal A token representing a constant value directly written in source code: number literals (`123` , `3.14`), string literals (`"hello"`), boolean literals (`true` , `false`), and `nil` . The scanner extracts the runtime value (Java `Double` , `String` , `Boolean` , or `LoxNil`) and stores it in `Token.literal` .

Logical Operators The `and` and `or` operators that combine boolean expressions with short-circuit evaluation. `and` returns the left operand if it's falsy (short-circuit), otherwise evaluates and returns the right operand. `or` returns the left operand if it's truthy (short-circuit), otherwise evaluates and returns the right operand.

Lox The programming language being implemented in this project—a small, dynamically-typed, object-oriented language with C-like syntax, designed by Bob Nystrom for educational purposes in "Crafting Interpreters."

LoxCallable A Java interface implemented by all callable entities in Lox: functions (`LoxFunction`), classes (`LoxClass`), and potentially native functions (`NativeCallable`). Defines `call()` and `arity()` methods that the interpreter uses uniformly.

LoxClass The runtime representation of a class, created when a class declaration is evaluated. Stores the class `name` , its `methods` (including `init`), and optional `superclass` . When called as a function, creates a new `LoxInstance` and invokes the initializer.

LoxFunction The runtime representation of a function, created when a function declaration is evaluated or a lambda is created. Stores the function's AST node (`declaration`), its `closure` (defining environment), and an `isInitializer` flag for distinguishing `init` methods.

LoxInstance The runtime representation of a class instance, created by calling a `LoxClass` . Stores a reference to its `klass` (for method lookup) and a map of `fields` (instance-specific data). Provides `get()` and `set()` methods for property access.

LoxNil The runtime representation of Lox's `nil` value, a singleton object indicating absence of meaningful value. Used as the default initializer value for variable declarations without explicit initializers and as the implicit return value from functions without `return` statements.

M

Mental Model An intuitive analogy or conceptual framework that helps understand a complex system before delving into technical details. Used throughout this design document to bridge from "what it feels like" to "how it actually works" for each component.

Method A function defined within a class body, implicitly bound to instances of that class. When called on an instance, the `this` keyword within the method body refers to that instance. Methods are stored in the `LoxClass.methods` map and accessed via dot notation.

Method Caching An optimization where the result of method lookup (resolving a method name to a specific `LoxFunction` in the class hierarchy) is cached at the call site, avoiding repeated traversal of the inheritance chain on subsequent calls with the same receiver type.

Method Resolution Order The order in which classes are searched when looking up a method on an instance. In Lox's single inheritance: first check the instance's class, then its superclass, then the superclass's superclass, etc., until finding the method or reaching the top (causing a runtime error).

Module A self-contained unit of code with its own environment, potentially exporting some definitions for use by importing code. While not in base Lox, extensions might add modules to support larger programs and code organization, implemented as `LoxModule` runtime objects.

N

Native Function A function implemented in the host language (Java) rather than Lox, exposed to Lox programs via the FFI. Useful for providing I/O, mathematical functions, or other capabilities difficult or inefficient to implement in pure Lox.

Nil Lox's null/none value, written as the keyword `nil` . The only value of the `LoxNil` type, representing absence of meaningful value. In truthiness rules, `nil` is falsy.

Number Lox's numeric type, representing double-precision floating-point values (Java `Double`). Number literals can include optional decimal points (`3` , `3.14`). All arithmetic operations work on numbers, with runtime type checking to prevent operations on non-numbers.

O

Object In the context of Lox's runtime values, any value that can be stored in variables, passed as arguments, or returned from functions. Includes `LoxNumber` , `LoxString` , `LoxBoolean` , `LoxNil` , `LoxFunction` , `LoxClass` , `LoxInstance` , and potentially `LoxArray` or

`LoxModule`. Not to be confused with Java's `Object` class, though Lox values are represented as Java `Object` references in the interpreter.

Operator Precedence Rules that determine which operations are performed first in expressions with multiple operators. In Lox, from highest to lowest: grouping `()`, unary `! -`, multiplication/division `* /`, addition/subtraction `+ -`, comparison `< > <= >=`, equality `== !=`, logical `and or`. The parser implements this via a cascade of recursive methods.

Optimization Pass A transformation phase applied to the AST or bytecode to improve performance without changing program semantics. Examples include constant folding, dead code elimination, or inlining. Typically run after parsing but before execution.

P

Panic-Mode Recovery An error recovery strategy used by parsers where, upon encountering a syntax error, they discard tokens (panic) until reaching a known synchronization point (like a statement boundary), then continue parsing. This prevents cascading error reports from a single mistake.

Parameter A variable declared in a function definition that receives a value (argument) when the function is called. Parameters become local variables in the function's execution environment, initialized with the corresponding argument values.

Parser The component that performs syntactic analysis, consuming tokens from the scanner and building an Abstract Syntax Tree according to the language's grammar rules. Our design uses a recursive descent parser implemented in the `Parser` class.

Parsing Expression Grammar (PEG) A formal grammar notation that describes a top-down parser's recognition process, well-suited for recursive descent parsers. Our expression parsing methods directly correspond to PEG rules for each precedence level.

Pretty Printer See **AST Printer**.

Primary Expression The most fundamental expressions that serve as building blocks for more complex expressions: literals, identifiers (variables), grouping parentheses, and other atomic forms. Parsed by the `primary()` method in the recursive descent parser.

Print Statement A statement that evaluates an expression and outputs its string representation to standard output, followed by a newline. The `Print` AST node contains the `expression` to evaluate. Used for program output and debugging.

Property A field or method accessible on a class instance via dot notation (`instance.property`). The interpreter handles property access uniformly: first check instance fields, then class methods (including inherited ones), raising a runtime error if neither exists.

R

Recursive Descent Parsing A top-down parsing technique where each nonterminal in the grammar is implemented as a function (method) that recursively calls other nonterminal functions. Our parser uses this approach, with methods like `expression()`, `equality()`, `comparison()`, etc., matching the precedence hierarchy.

Register VM A virtual machine architecture that uses named registers (slots) rather than an operand stack for holding intermediate values. Typically faster but more complex to compile to than stack VMs. Contrasts with the stack VM approach mentioned in future extensions.

REPL (Read-Eval-Print Loop) An interactive programming environment that reads input, evaluates it, prints the result, and loops. While not a goal for the base project, a `Repl` class could be added to allow experimenting with Lox code without writing files.

Return Statement A statement that immediately exits the current function, optionally providing a value that becomes the function call's result. The `Return` AST node contains the `keyword` token (for error location) and optional `value` expression. Implemented via a `Return` exception that unwinds to the call site.

Runtime Error An error that occurs during program execution (interpretation), such as type mismatches, undefined variables, division by zero, or calling non-functions. Represented by the `RuntimeError` exception class with `token` (location) and `message` fields. The interpreter catches these and reports them to the user.

Runtime Value The concrete data that exists during program execution, representing the result of evaluating expressions. In our implementation, runtime values are Java objects (`Double`, `String`, `Boolean`, `LoxNil`, `LoxFunction`, `LoxClass`, `LoxInstance`) stored in variables and manipulated by operations.

S

Scanner The component that performs lexical analysis, converting source code (string) into a list of tokens. Also called a lexer. Implemented by the `Scanner` class with methods to recognize tokens by scanning characters.

Semantic Analysis The process of interpreting the meaning of valid syntactic structures, including type checking, variable resolution, and other static validations. In our simple interpreter, much of semantic analysis happens at runtime (dynamic checking), though some occurs during parsing (variable resolution for closures).

Short-Circuit Evaluation An optimization where the right operand of a logical operator (`and` , `or`) is evaluated only if necessary to determine the result. For `and` , if the left operand is falsy, the result is the left operand (right not evaluated). For `or` , if the left operand is truthy, the result is the left operand.

S-Expression A parenthesized notation for representing tree structures, commonly used in Lisp and for debugging ASTs. Example: `(* (+ 1 2) 3)` represents the multiplication of `(+ 1 2)` and `3` . The `AstPrinter` produces S-expressions from ASTs.

Stack VM A virtual machine architecture that uses an operand stack for holding intermediate values and executing instructions (push, pop, operate). Simpler to compile to than register VMs and conceptually similar to the tree-walking evaluator's implicit call stack.

Standard Library A collection of built-in functions and types available to all Lox programs without explicit import. While not a goal for the base project, extensions might add a standard library with I/O, mathematical, and utility functions via native functions.

Statement A piece of syntax that performs an action but doesn't produce a value (unlike expressions). Examples include variable declarations, print statements, conditionals, loops, and function declarations. Represented by the `Stmt` abstract class and its concrete subclasses.

String A sequence of characters, created with double quotes (`"hello"`) and supporting escape sequences (`\n` , `\t` , `\"` , etc.). Strings can be concatenated with `+` operator (with any non-string operand converted via `stringify()`). Represented at runtime as Java `String` objects.

String Interning An optimization where identical strings are stored once in a shared pool and reused, saving memory and enabling fast equality comparison via reference equality. Could be applied to Lox string literals and runtime string values.

Super A keyword used inside subclass methods to call a method from the superclass, bypassing any override in the subclass. The `Super` AST node stores the `keyword` token and `method` identifier. The interpreter resolves this by looking up the method in the superclass with `this` bound to the current instance.

Switch Statement A multi-way conditional statement that compares a value against multiple cases. While not in base Lox, extensions might add `switch` / `case` / `default` statements as syntactic sugar for nested `if` / `else if` chains.

Synchronization Point A token where the parser can safely resume after error recovery in panic mode. Typically statement boundaries (semicolons, closing braces, or keywords like `var` , `fun` , `class` , `for` , `if` , `while` , `return`). The `Parser.synchronize()` method skips tokens until reaching one.

Syntactic Analysis The process of analyzing a sequence of tokens to determine its grammatical structure according to the language's grammar rules, producing an AST. Performed by the parser.

T

Tail Call Optimization An optimization where a function call in tail position (the last action before returning) reuses the current stack frame instead of allocating a new one, preventing stack overflow in deeply recursive algorithms. Could be added to the interpreter for functional programming patterns.

This A keyword referring to the current instance within a method body. The `This` AST node stores the `keyword` token. During method calls, the interpreter binds `this` to the receiving instance by creating a special environment that defines `this` before executing the method body.

Token A categorized lexical unit extracted from source code by the scanner. Represented by the `Token` class with fields: `type` (a `TokenType` enum), `lexeme` (raw text), `literal` (runtime value for literals), and `line` (source line number). Tokens are the parser's input.

Token Stream Visualizer A debugging utility that prints the list of tokens produced by the scanner, showing type, lexeme, and line numbers. Helps verify the scanner is correctly recognizing tokens before parsing.

TokenType An enumeration of all possible token categories in Lox, including punctuation (`LEFT_PAREN` , `COMMA`), operators (`PLUS` , `EQUAL_EQUAL`), keywords (`VAR` , `FUN`), literals (`NUMBER` , `STRING`), and special tokens (`IDENTIFIER` , `EOF`).

Tree-Walking Interpreter An interpreter that executes code by recursively traversing the Abstract Syntax Tree, evaluating each node according to its semantics. Our design uses this approach, contrasting with bytecode interpreters that compile to an intermediate representation first.

Truthiness The boolean interpretation of values in conditional contexts (if conditions, while conditions, logical operators). In Lox: `false` and `nil` are falsy; everything else (including `0`, empty string `""`, and `true`) is truthy. The `isTruthy()` method implements this rule.

U

Unary Expression An expression with a single operator and operand, such as `-5` or `!true`. Represented by the `Unary` AST node with `operator` and `right` fields. The parser recognizes unary operators at a higher precedence level than binary operators.

Upvalue A variable captured from an enclosing scope by a closure. In more advanced implementations, upvalues are specially allocated cells that survive after their declaring function returns, allowing closures to access them. Our design simply captures the entire environment.

V

Variable A named storage location that holds a value, declared with `var` statements. Variables can be accessed by name (via `Variable` expression node) and reassigned (via `Assign` expression node). Lookup proceeds through the environment chain following lexical scoping rules.

Variable Declaration A statement that introduces a new variable into the current scope, optionally with an initializer expression: `var x = 5;` or `var x;` (initialized to `nil`). The `Var` AST node stores the `name` token and optional `initializer` expression.

Virtual Machine (VM) An abstract computing machine that executes bytecode instructions, typically implemented in software. In interpreter architectures, a VM provides a faster execution model than tree-walking by compiling to a compact bytecode and using efficient dispatch loops.

Visitor Pattern A behavioral design pattern that separates operations from the object structure they operate on. In our AST design, the `Expr.Visitor<R>` and `Stmt.Visitor<V>` interfaces define operations, and each AST node's `accept()` method calls the appropriate `visitXxx()` method on a visitor, enabling double dispatch without `instanceof` checks.

W

While Loop A control flow statement that repeatedly executes a body statement as long as a condition expression evaluates to truthy: `while (condition) body`. Represented by the `While` AST node with `condition` and `body` fields. The interpreter evaluates the condition before each iteration.

Whitespace Characters that separate tokens but carry no meaning: spaces, tabs, carriage returns, and newlines. The scanner skips whitespace between tokens. Newlines increment the line counter for accurate error reporting.