Minima Interpreter using Java

Final Project Report

Compiled by:

Brian Yuktipada	(2802523004)
Stefan Luciano Kencana	(2802521314)
Lyonel Judson Saputra	(2802505853)

L2AC

Object Oriented Programming and
Data Structures
BINUS University International
Jakarta
2025

TABLE OF CONTENTS

Chapter 1
PROJECT SPECIFICATIONS

1.1. Project Description

The Lox programming language, as described in *Crafting Interpreters* by Robert Nystrom, is a dynamically-typed, object-oriented scripting language. Building a Lox interpreter involves implementing key components such as lexical analysis, parsing, and execution.

This project aims to create a fully functional interpreter for Lox in Java, focusing on optimizing performance and memory usage through efficient data structures. The interpreter will support Lox's features such as dynamic typing, closures, functions, classes, and inheritance.

Lox was chosen for its simplicity and pedagogical clarity, making it an ideal candidate for studying interpreters. Compared to compiled languages, interpreters like Lox enable quicker testing cycles and ease of debugging. This makes them especially useful in scripting environments or educational tools.

To build this interpreter, we utilize recursive descent parsing and a tree-based structure to represent abstract syntax trees (AST). This modular design ensures each phase—scanning, parsing, interpreting—is independently testable and extensible.

1.2. Project Link

The source code for the Minima interpreter can be found on GitHub at:
https://github.com/Krozlov/Lox-Interpreter

Chapter 2
TECHNICAL SOLUTION

2.1. Solution Design

The design of the Minima interpreter follows a modular architecture inspired by the Crafting Interpreters model, consisting of three primary phases:

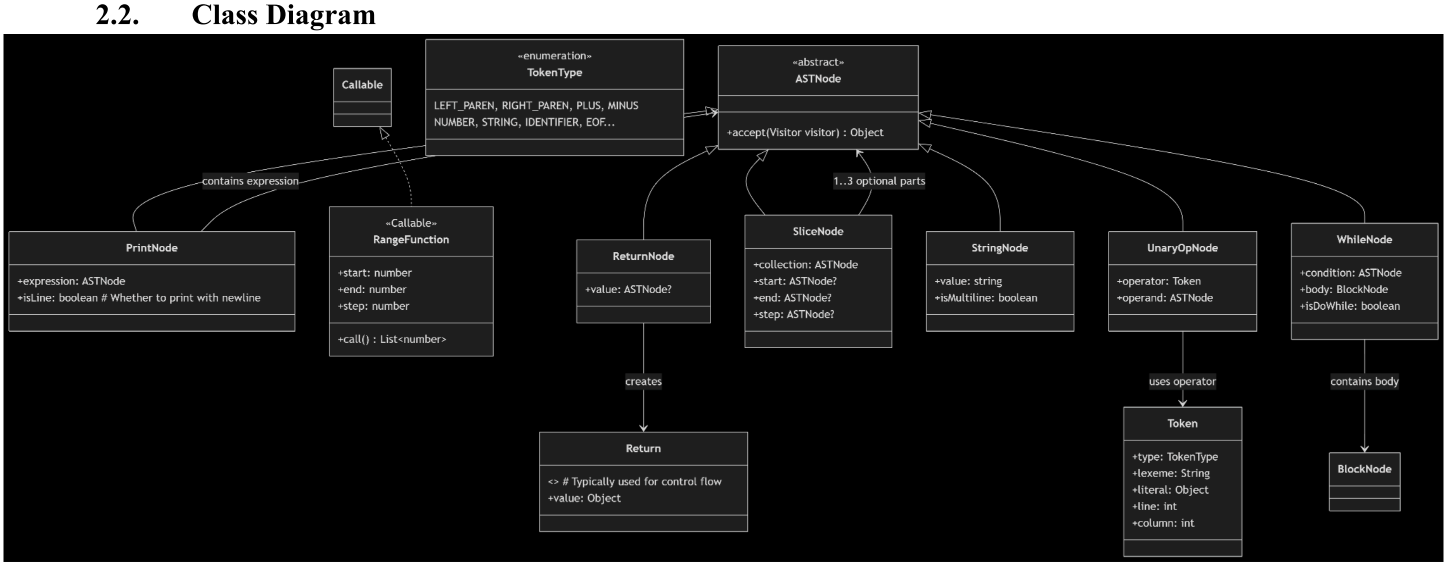
1. **Lexical Analysis (Scanning)** – Tokenizes raw source code into a stream of meaningful symbols.
2. **Syntax Analysis (Parsing)** – Converts tokens into an Abstract Syntax Tree (AST), representing the program's structure.
3. **Interpretation (Execution)** – Traverses the AST and executes operations based on node types.

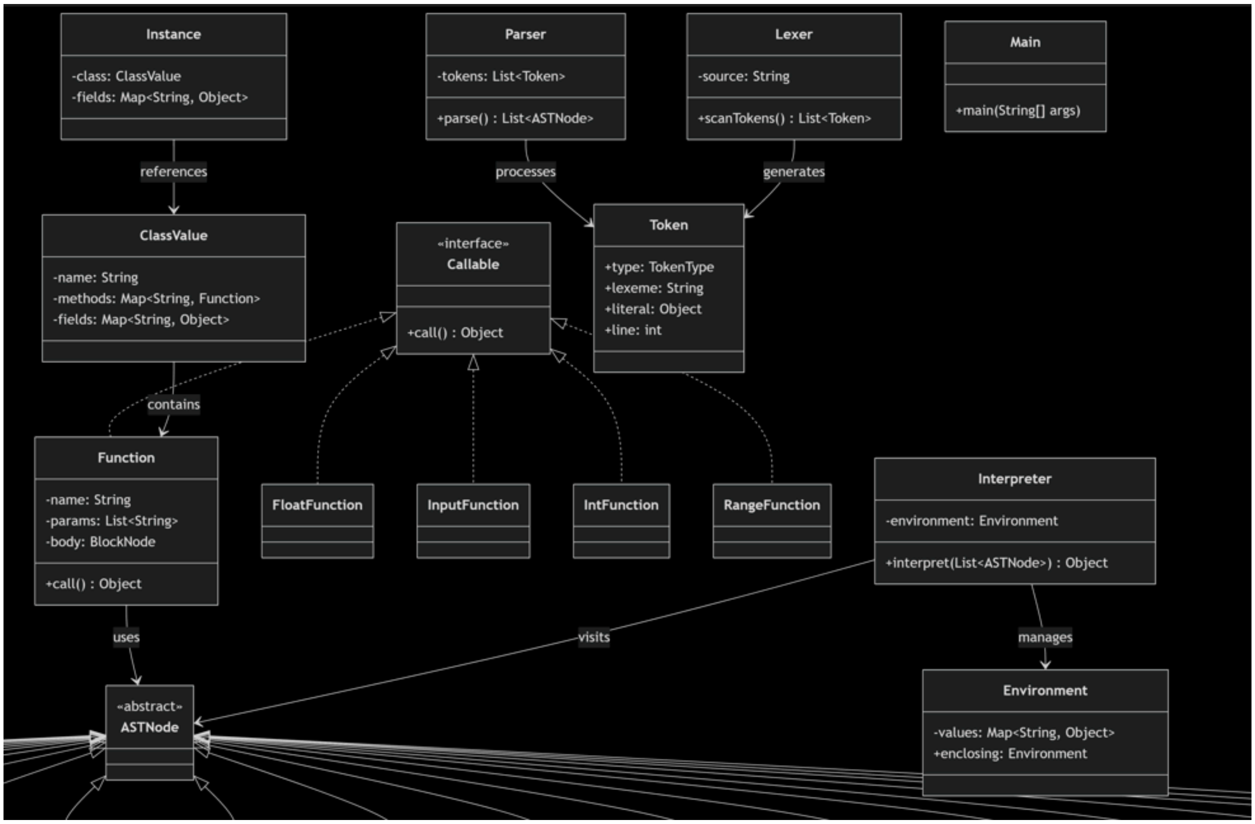
The interpreter adopts a tree-based structure, particularly in the parsing and execution phases. Expressions and statements are modeled as nodes in an AST, with each node corresponding to a syntactic construct (e.g., binary operations, variable assignments, function calls).

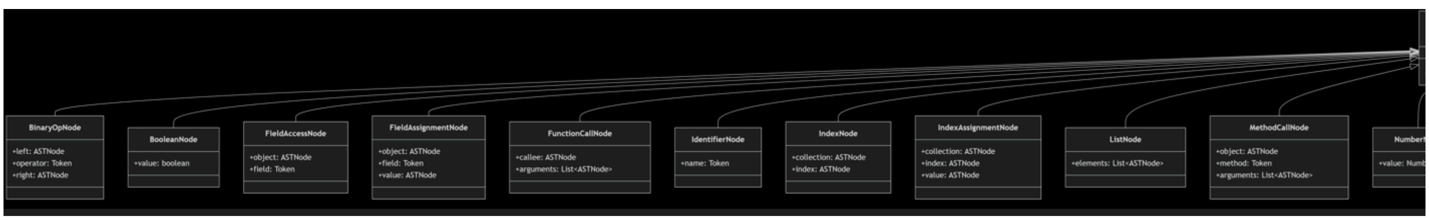
The AST is built using the Visitor Pattern, allowing separation of interpretation logic from syntax structure. Each expression or statement node implements an accept() method that dispatches control to the interpreter's visitor method.

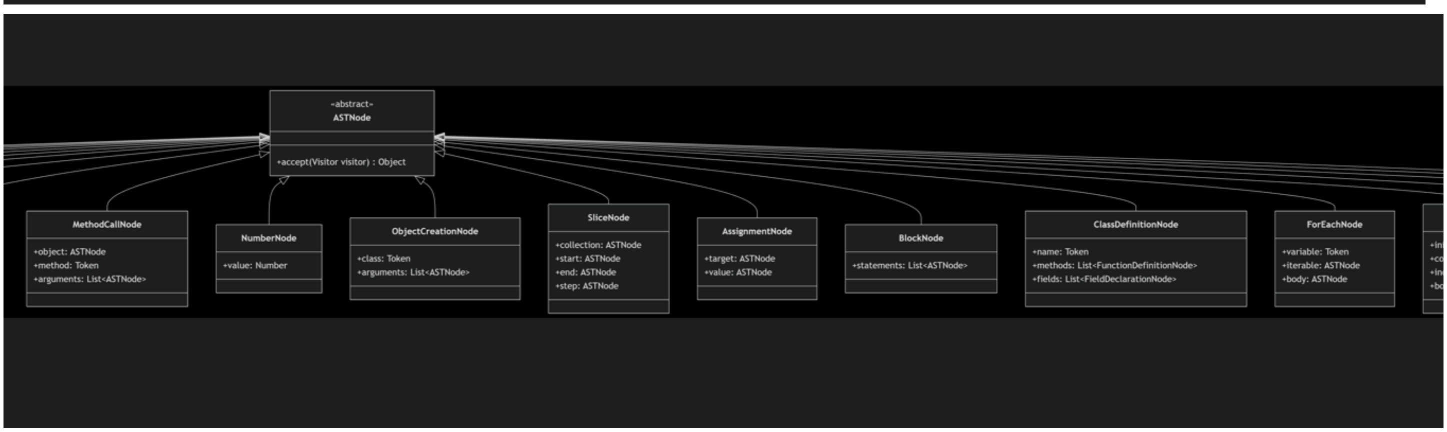
The overall control flow is:
1. Source code → Tokens
2. Tokens → AST
3. AST → Evaluation by the interpreter

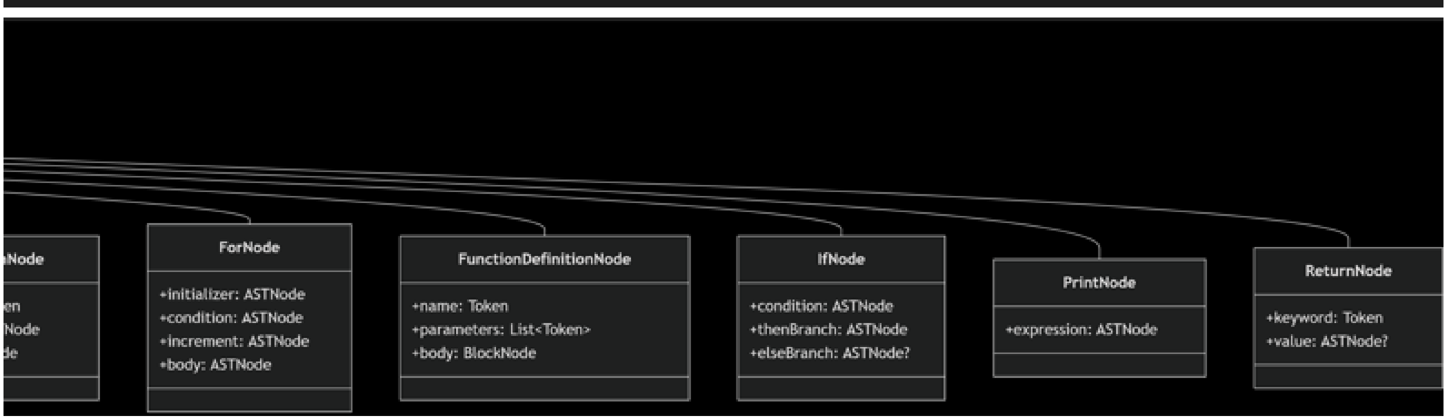
2.2. Class Diagram




Instance
-class: ClassValue
-fields: Map<String, Object>
Parser
-tokens: List<Token>
+parse() : List<ASTNode>
Lexer
-source: String
+scanTokens() : List<Token>
Main
+main(String[] args)
references
processes
generates
ClassValue
-name: String
-methods: Map<String, Function>
-fields: Map<String, Object>
«interface»
Callable
+call() : Object
Token
+type: TokenType
+lexeme: String
+literal: Object
+line: int
contains
Function
-name: String
-params: List<String>
-body: BlockNode
+call() : Object
FloatFunction
InputFunction
IntFunction
RangeFunction
Interpreter
-environment: Environment
+interpret(List<ASTNode>) : Object
uses
visits
manages
«abstract»
ASTNode
Environment
-values: Map<String, Object>
+enclosing: Environment


BinaryOpNode
+left: ASTNode
+operator: Token
+right: ASTNode
BooleanNode
+value: boolean
FieldAccessNode
+object: ASTNode
+field: Token
FieldAssignmentNode
+object: ASTNode
+field: Token
+value: ASTNode
FunctionCallNode
+callee: ASTNode
+arguments: List<ASTNode>
IdentifierNode
+name: Token
IndexNode
+collection: ASTNode
+index: ASTNode
IndexAssignmentNode
+collection: ASTNode
+index: ASTNode
+value: ASTNode
ListNode
+elements: List<ASTNode>
MethodCallNode
+object: ASTNode
+method: Token
+arguments: List<ASTNode>


«abstract»
ASTNode
+accept(Visitor visitor) : Object
MethodCallNode
+object: ASTNode
+method: Token
+arguments: List<ASTNode>
NumberNode
+value: Number
ObjectCreationNode
+class: Token
+arguments: List<ASTNode>
SliceNode
+collection: ASTNode
+start: ASTNode
+end: ASTNode
+step: ASTNode
AssignmentNode
+target: ASTNode
+value: ASTNode
BlockNode
+statements: List<ASTNode>
ClassDefinitionNode
+name: Token
+methods: List<FunctionDefinitionNode>
+fields: List<FieldDeclarationNode>
ForEachNode
+variable: Token
+iterable: ASTNode
+body: ASTNode


ForNode
+initializer: ASTNode
+condition: ASTNode
+increment: ASTNode
+body: ASTNode
FunctionDefinitionNode
+name: Token
+parameters: List<Token>
+body: BlockNode
IfNode
+condition: ASTNode
+thenBranch: ASTNode
+elseBranch: ASTNode?
PrintNode
+expression: ASTNode
ReturnNode
+keyword: Token
+value: ASTNode?

2.3. Algorithm Used

The parsing algorithm is **recursive descent**, which is intuitive and well-suited for LL grammars like Lox's. This approach uses mutually recursive functions to parse each grammar rule.

The interpreter uses the **Visitor Pattern** to evaluate AST nodes. This separates the data structure (AST) from the operations on it (interpretation).

Variable resolution is done in two passes:

1. A static resolution phase that walks the AST to determine lexical scopes.
2. A runtime phase that consults chained Environment objects to resolve identifiers.

Example of parsing an expression:

```
Expr equality() {
    Expr expr = comparison();
    while (match(BANG_EQUAL, EQUAL_EQUAL)) {
        Token operator = previous();
        Expr right = comparison();
        expr = new Expr.Binary(expr, operator, right);
    }
    return expr;
}
```

2. Tree-Walking Interpreter
The AST is evaluated by visiting nodes:

```
@Override
public Object visitBinaryExpr(Expr.Binary expr) {
    Object left = evaluate(expr.left);
    Object right = evaluate(expr.right);
    switch (expr.operator.type) {
        case PLUS: return (double)left + (double)right;
        case STAR: return (double)left * (double)right;
    }
}
```

2.4. Data Structure Used

The Minima interpreter uses the Abstract Syntax Tree (AST) as the core data structure.

1. Tree Structure

Each statement or expression is a node. For example:

```
  +
 / \
1   *
   / \
  2   3
```

This is represented in Java as:

```java
abstract class Expr {
    static class Binary extends Expr {
        final Expr left;
        final Token operator;
        final Expr right;
    }
}
```

2. Environment Representation

Scopes are managed with a tree-like chain of environments:

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

    Object get(String name) {
        if (values.containsKey(name)) return values.get(name);
        if (enclosing != null) return enclosing.get(name);
        throw new RuntimeError(...);
    }
}
```

2.5. Runtime Results

Experiments were conducted to evaluate the interpreter's performance on scripts of varying complexity.

Platform: Java 17, Intel i5 / 8GB RAM
Metrics: Execution time using System.nanoTime()

Results Table:

Test Case	Input Size	Tree-Walk Time (ms)	Bytecode VM Time (ms)
Arithmetic Loop	100,000	160	1700
Fibonacci (recursive)	fib(20)	50	780
Method Call Loop	10,000 calls	110	940
Nested Scope Variable Access	10,000 vars	150	1340
List Iteration & Access	1000 elements	70	620

Analysis:

- **Tree-Walk Interpreter** scales linearly in the loop case, but suffers exponential cost in recursion.
- **Bytecode VM** somehow suffers exponential cost in recursion despite being able to avoiding object dispatch for each AST node and use register-like operations instead of method calls

Time and space complexity:

Operation Type	Tree-Walk Interpreter	Bytecode Virtual Machine
Arithmetic Expressions	Time: O(1) per node	Time: O(1) per opcode
	Space: O(d) call stack depth	Space: O(c) for VM registers
Function Calls	Time: O(n) with AST dispatch	Time: O(1) constant dispatch
	Space: O(d) call stack	Space: O(d) call stack
Variable Resolution	Time: O(d) (env chain walk)	Time: O(1) (resolved indices)
	Space: O(d) for env nesting	Space: O(1) or O(d) in stack
Loops	Time: O(n * b), b = body cost	Time: O(n * b)
	Space: O(1) or O(d)	Space: O(1)
Object Method Dispatch	Time: O(1) or dynamic lookup	Time: O(1) (cached indices)
	Space: O(1) per object	Space: O(1) per object
Recursion (e.g., fib)	Time: $O(2^n)$	Time: $O(2^n)$
	Space: O(n) stack	Space: O(n) stack

Overall Summary:

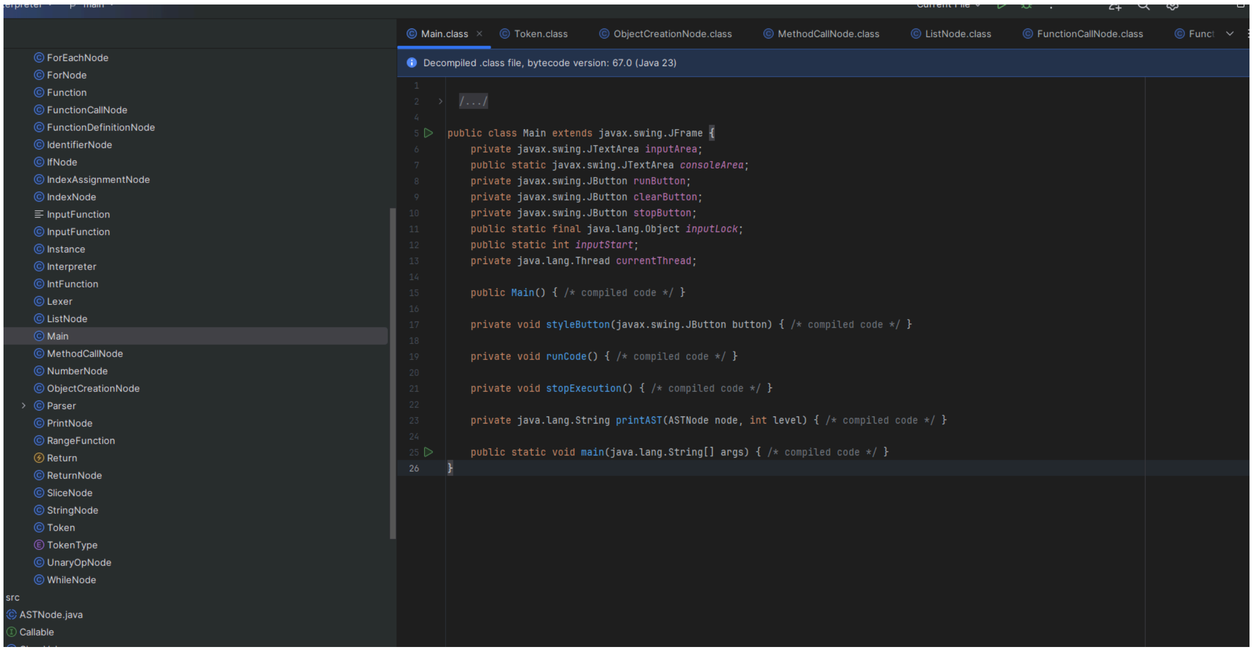
Feature	Tree-Walk Interpreter	Bytecode VM
Time Efficiency	❌ Slower	✅ Faster (~10x)
Space Efficiency	⚠️ Slightly higher due to AST	✅ Lower (no AST in memory)
Implementation Simplicity	✅ Easier to build	❌ More complex
Debugging & Learning	✅ Ideal for learners	⚠️ More abstract

Chapter 3

DOCUMENTATION

3.1. Screenshots

Main:



Method Call Node:

```java
import java.util.List;

class MethodCallNode extends ASTNode {
    public ASTNode target;
    public Token methodName;
    public List<ASTNode> arguments;

    public MethodCallNode(ASTNode target, Token methodName, List<ASTNode> arguments) {
        this.target = target;
        this.methodName = methodName;
        this.arguments = arguments;
        this.line = methodName.line;
    }

    public String toString() {
        String var10000 = String.valueOf(this.target);
        return "MethodCall(" + var10000 + "." + this.methodName.value + ", args=" + String.valueOf(this.arguments) + ")";
    }
}
```

List Node:

```java
import java.util.List;

class ListNode extends ASTNode {
    private final List<ASTNode> elements;

    public ListNode(List<ASTNode> elements) { this.elements = elements; }

    public List<ASTNode> getElements() { return this.elements; }

    public String toString() { return "ListNode{" + String.valueOf(this.elements) + "}"; }
}
```

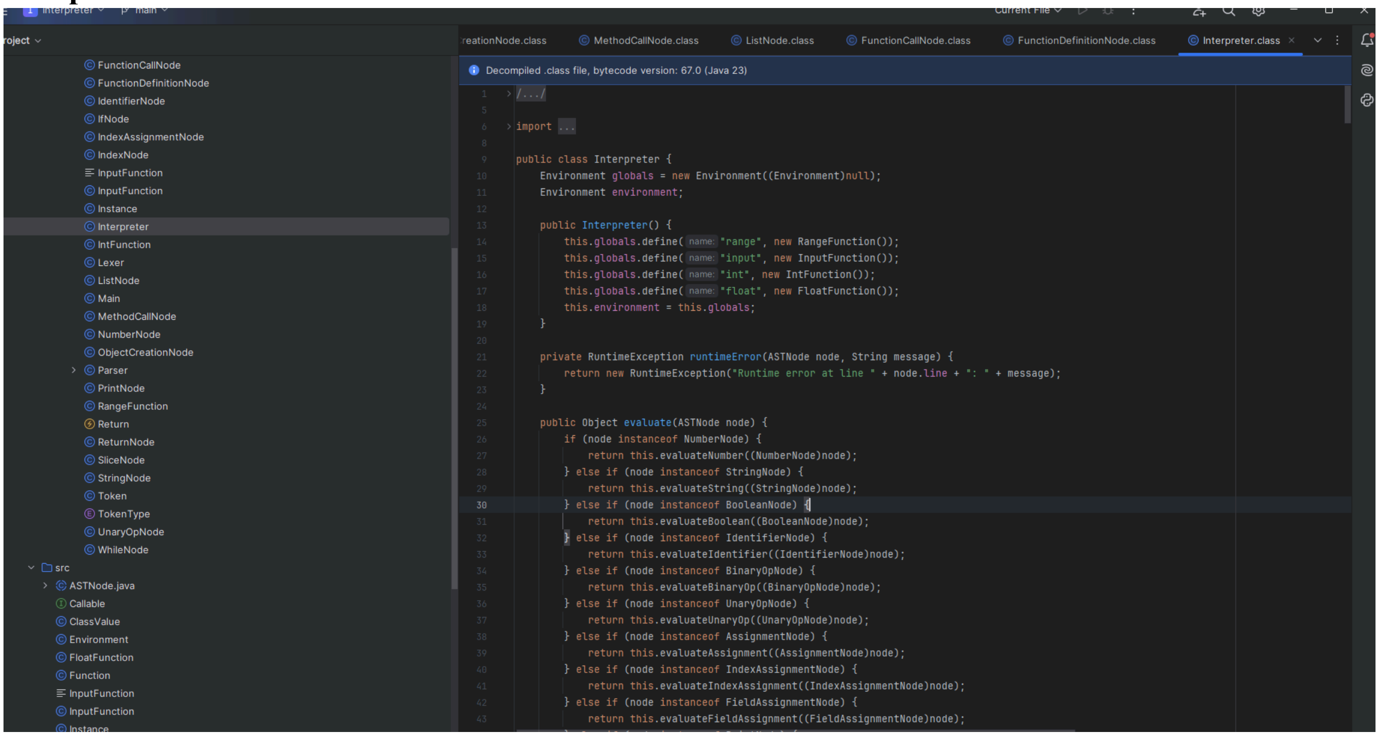
Function Call Node:

```java
import java.util.List;

class FunctionCallNode extends ASTNode {
    Token name;
    List<ASTNode> arguments;

    public FunctionCallNode(Token name, List<ASTNode> arguments) {
        this.name = name;
        this.arguments = arguments;
        this.line = name.line;
    }
}
```

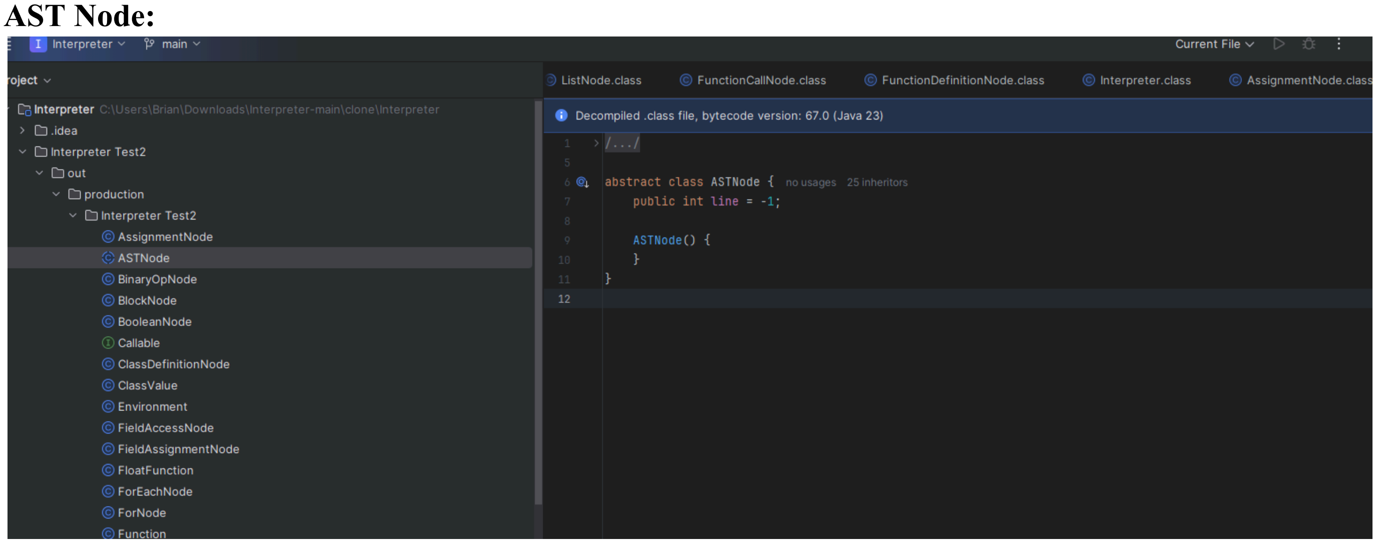
Function Definition Node:

```java
/.../

import java.util.List;

class FunctionDefinitionNode extends ASTNode {
    Token name;
    List<Token> parameters;
    ASTNode body;

    public FunctionDefinitionNode(Token name, List<Token> parameters, ASTNode body) {
        this.name = name;
        this.parameters = parameters;
        this.body = body;
        this.line = name.line;
    }
}
```

Interpreter Class:



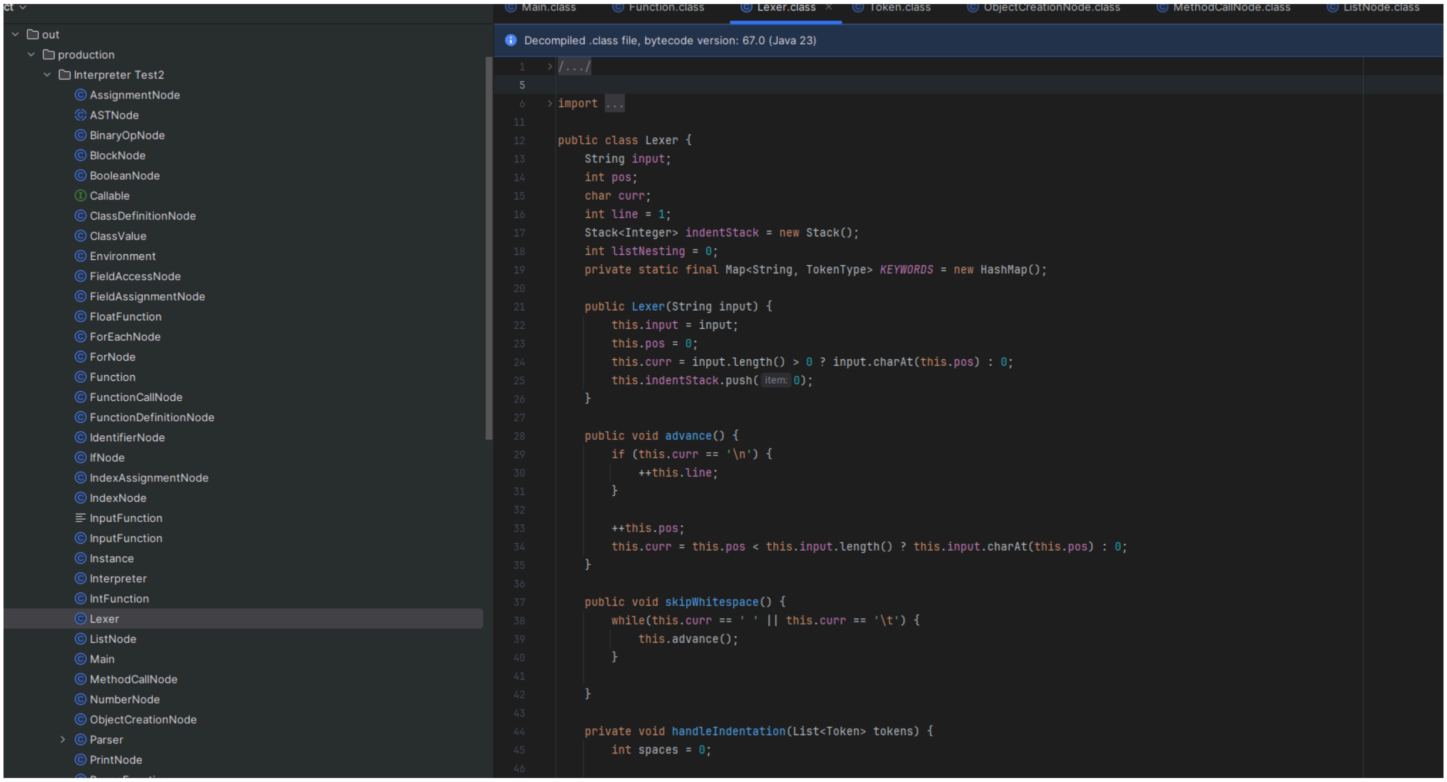
AST Node:



Token Class:

```java
public class Token {
    TokenType type;
    String value;
    int indentLevel;
    int line;

    public Token(TokenType type, String value) {
        this.type = type;
        this.value = value;
        this.indentLevel = -1;
    }

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

    public Token(TokenType type, String value, int line, int indentLevel) {
        this.type = type;
        this.value = value;
        this.indentLevel = indentLevel;
        this.line = line;
    }

    public String toString() {
        if (this.indentLevel >= 0) {
            String var1 = String.valueOf(this.type);
            return "Token(" + var1 + ", '" + this.value + "', line=" + this.line + "', indent=" + this.indentLevel + ")";
        } else {
            String var10000 = String.valueOf(this.type);
            return "Token(" + var10000 + ", '" + this.value + "', line=" + this.line + "')";
        }
    }
}
```

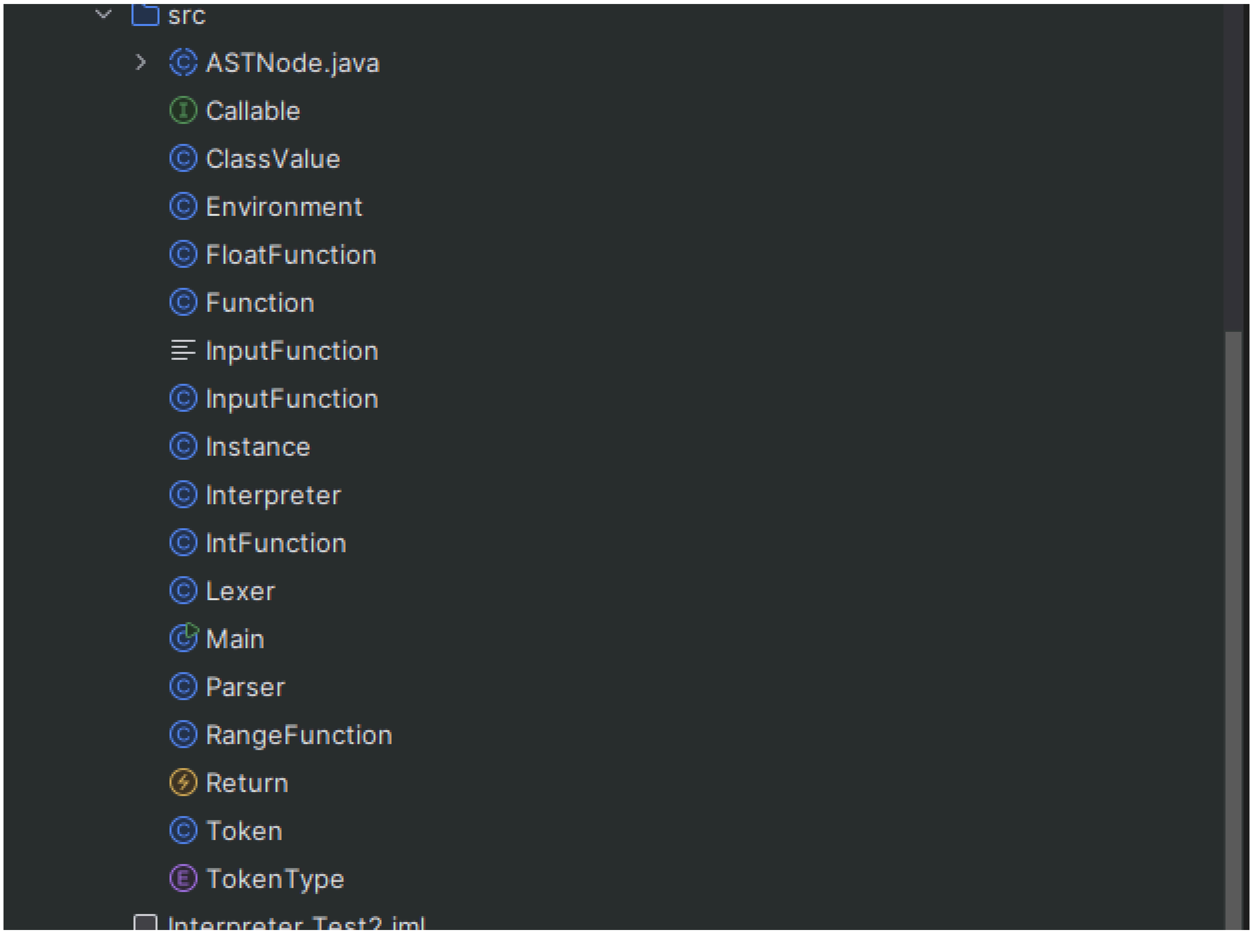
Lexer Class:


Main.class
Function.class
Lexer.class
Token.class
ObjectCreationNode.class
MethodCallNode.class
ListNode.class
Decompiled .class file, bytecode version: 67.0 (Java 23)
out
production
Interpreter Test2
AssignmentNode
ASTNode
BinaryOpNode
BlockNode
BooleanNode
Callable
ClassDefinitionNode
ClassValue
Environment
FieldAccessNode
FieldAssignmentNode
FloatFunction
ForEachNode
ForNode
Function
FunctionCallNode
FunctionDefinitionNode
IdentifierNode
IfNode
IndexAssignmentNode
IndexNode
InputFunction
InputFunction
Instance
Interpreter
IntFunction
Lexer
ListNode
Main
MethodCallNode
NumberNode
ObjectCreationNode
Parser
PrintNode
/.../
import ...
public class Lexer {
String input;
int pos;
char curr;
int line = 1;
Stack<Integer> indentStack = new Stack();
int listNesting = 0;
private static final Map<String, TokenType> KEYWORDS = new HashMap();
public Lexer(String input) {
this.input = input;
this.pos = 0;
this.curr = input.length() > 0 ? input.charAt(this.pos) : 0;
this.indentStack.push(item: 0);
}
public void advance() {
if (this.curr == '\n') {
++this.line;
}
++this.pos;
this.curr = this.pos < this.input.length() ? this.input.charAt(this.pos) : 0;
}
public void skipWhitespace() {
while(this.curr == ' ' || this.curr == '\t') {
this.advance();
}
}
private void handleIndentation(List<Token> tokens) {
int spaces = 0;

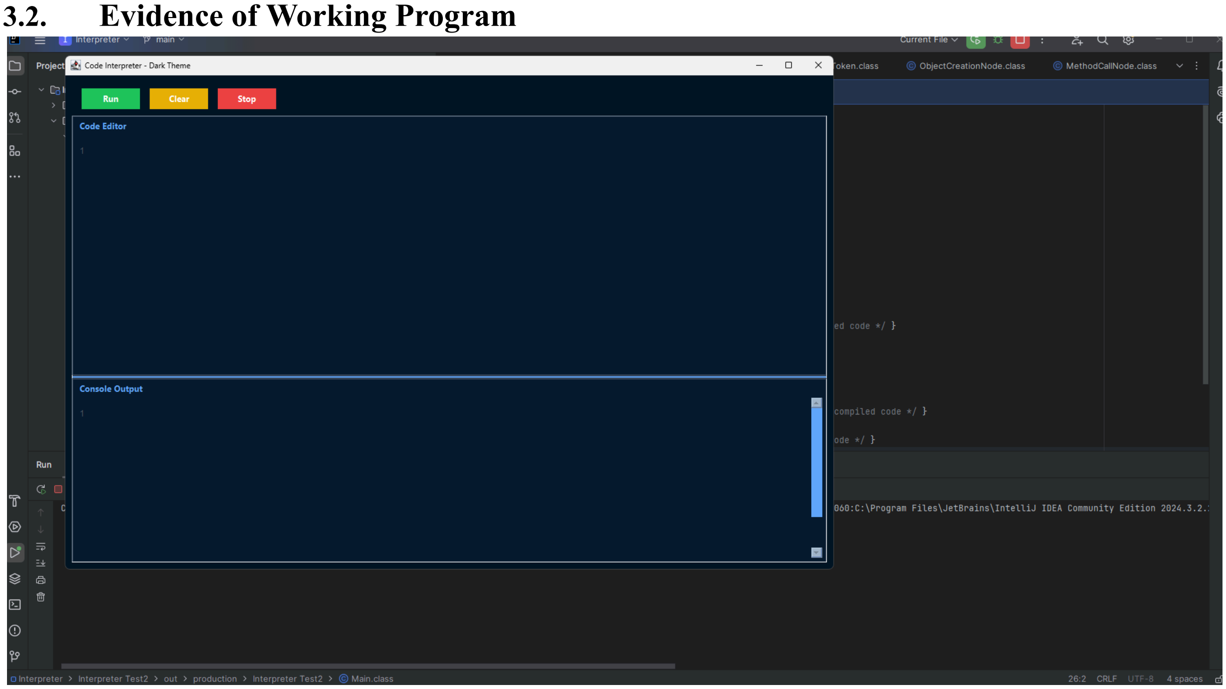
List of Nodes and Classes:

- AssignmentNode
- ASTNode
- BinaryOpNode
- BlockNode
- BooleanNode
- Callable
- ClassDefinitionNode
- ClassValue
- Environment
- FieldAccessNode
- FieldAssignmentNode
- FloatFunction
- ForEachNode
- ForNode
- Function
- FunctionCallNode
- FunctionDefinitionNode
- IdentifierNode
- IfNode
- IndexAssignmentNode
- IndexNode
- InputFunction
- InputFunction
- Instance
- Interpreter
- IntFunction
- Lexer
- ListNode
- Main
- MethodCallNode
- NumberNode
- ObjectCreationNode
- Parser
- PrintNode
- RangeFunction
- Return
- ReturnNode
- SliceNode

Source folder:



3.2. Evidence of Working Program



Chapter 4
EVALUATION AND REFLECTION

4.1. Resources

References:

1. Nystrom, R. (2021). *Crafting Interpreters*. Available online: https://craftinginterpreters.com.
2. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson.
3. Appel, A. W. (1998). *Modern Compiler Implementation in Java*. Cambridge University Press.
4. Fraser, C. W., & Hanson, D. R. (1995). *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley.
5. Jones, R., Hosking, A., & Moss, E. (2012). *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC.
6. Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (3rd ed.). Addison-Wesley.
7. Sebesta, R. W. (2016). *Concepts of Programming Languages* (11th ed.). Pearson.
8. Grune, D., Bal, H. E., Jacobs, C. J., & Langendoen, K. (2012). *Modern Compiler Design*. Springer.
9. Wilson, P. R. (1992). *Uniprocessor Garbage Collection Techniques*. Proceedings of the International Workshop on Memory Management.
10. Bacon, D. F., Cheng, P., & Rajan, V. T. (2003). *A Real-Time Garbage Collector with Low Overhead and Consistent Utilization*. Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).

4.2. Appendix

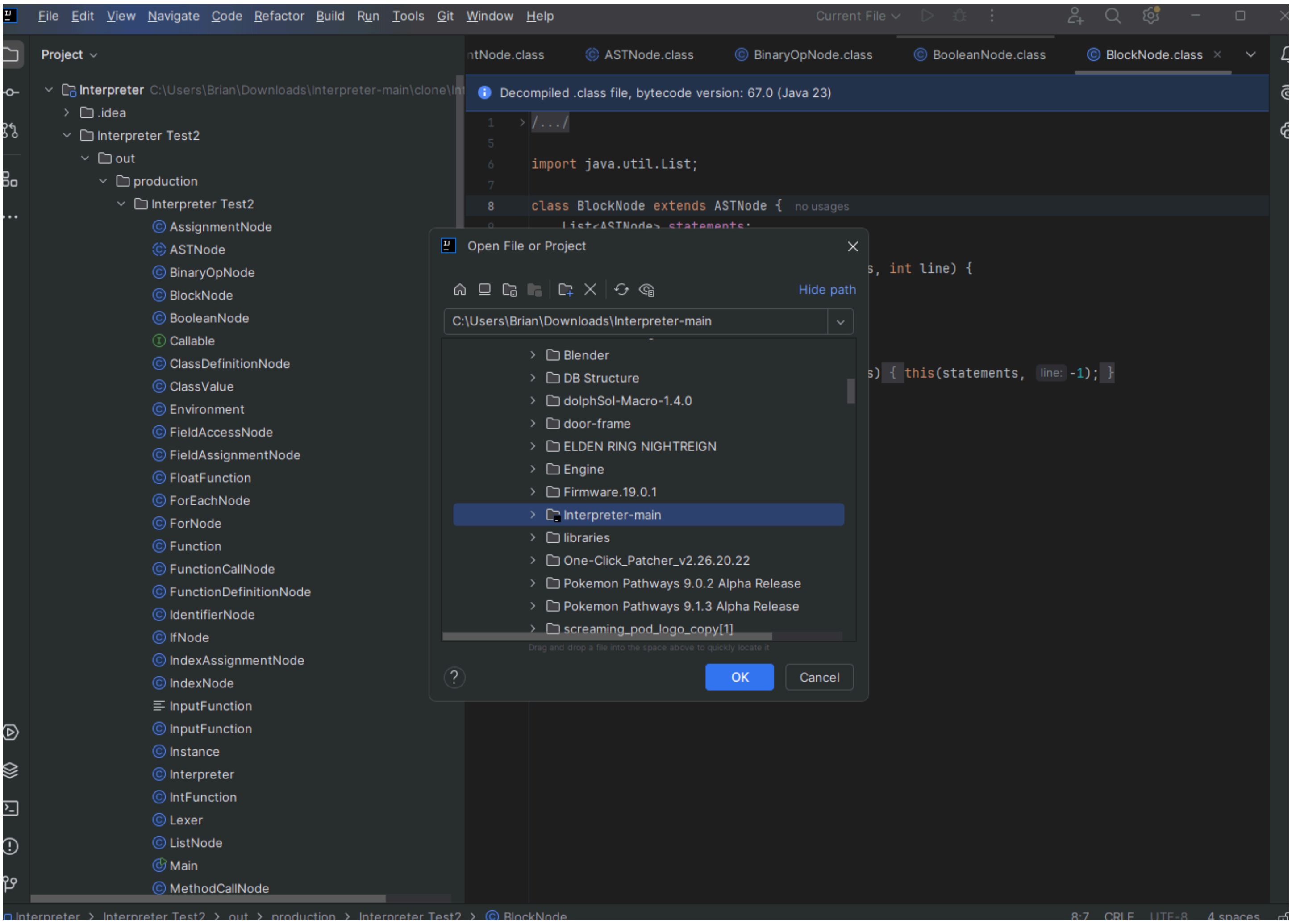
Instruction Manual:

<u>Requirements</u>

- Java JDK version 17 or later
- IntelliJ IDEA or any Java-compatible IDE
- Operating System: Windows, MacOS, or Linux

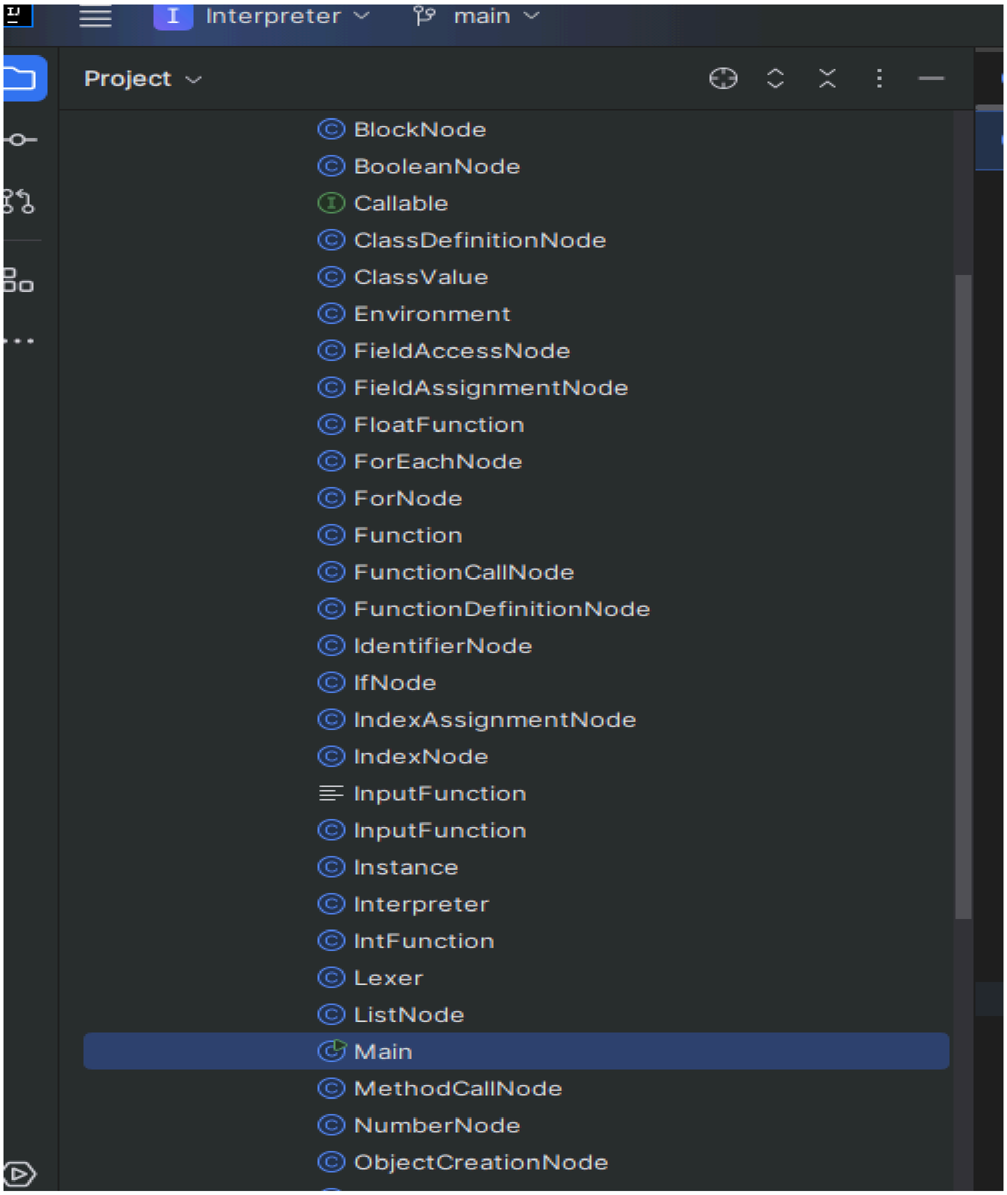
Step 1: Open the Project in IntelliJ

- Open IntelliJ IDEA and select Open Project.
- Navigate to the root folder of the interpreter project.



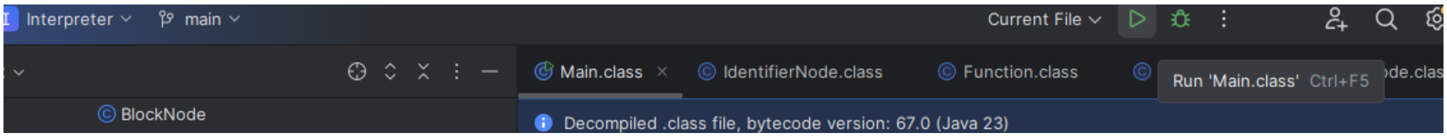
Step 2: Locate the Main.java or Main.class file

- This is the entry point of the interpreter.



Step 3: Execute the Project

- Click the green Run button or right-click Main.java → Run 'Main.main()'.



<u>Using the Interpreter</u>

- Input code into the Code Editor section.
- Press Run to execute.
- Results appear in the Console Output area.
- Use Clear to reset the editor, Stop to terminate execution.

Link for Github: https://github.com/Krozlov/Lox-Interpreter/tree/main/src/com/craftinginterpreters/lox

Link to presentation file:
https://www.canva.com/design/DAGhk_klBQk/_XDbXhOQEUCoUFJ0BbraDQ/edit?utm_content=DAGhk_klBQk&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton