DRAFT

Quick Guide to Build Your Own Simple Interpreter Using Modern C++

Prepared By Ayman Alheraki

First Edition

Quick Guide to Build Your Own Simple Interpreter in Modern C++

Prepared by Ayman Alheraki

ForgeVM.org

July 2025

Contents

Author's Preface

This quick guide was written with a sincere desire to benefit the followers of the ForgeVM project and help them enter one of the most fascinating and foundational fields in computer science: the design and implementation of programming languages.

Many passionate learners dream of building their own language, yet they often encounter technical or conceptual barriers along the way. This guide is a stepping stone to overcoming those obstacles. It simplifies the core concepts and walks the reader through the practical stages of building a minimal yet functional interpreter using Modern C++20/23.

While I am currently working on my own full-fledged programming language project—and preparing a comprehensive 750-page book that covers every detail of its design—I felt the need to also offer this concise, focused guide. Its purpose is to make the world of interpreters more accessible and to empower enthusiasts with the knowledge and confidence to begin their own journey.

I hope this guide brings value to readers and lights the path for those aiming to turn their language design dreams into reality.

Stay Connected

For more discussions and valuable content about **Quick Guide to Build Your Own Simple Interpreter in Modern C++**.

I invite you to follow me on **LinkedIn**:

https://linkedin.com/in/aymanalheraki

You can also visit my personal website:

https://simplifycpp.org

Ayman Alheraki

Chapter 1

Introduction

1.1 What Is an Interpreter?

An interpreter is a type of program that **reads source code** and **executes it directly**, without first compiling it to native machine code. This makes interpreters ideal for **dynamic**, **interactive**, and **lightweight languages** such as Python, Lua, and JavaScript.
Unlike compilers that generate an executable binary, interpreters work **at runtime**, analyzing and executing code line by line or statement by statement.

1.2 Key Characteristics of Interpreters:

Table 2-1: Interpreter vs Compiler Comparison

Feature	Interpreter	Compiler
Output	Executes directly	Generates executable binary
Runtime Speed	Slower (interpreted at runtime)	Faster (compiled beforehand)
Error Handling	Detects errors during execution	Detects errors before execution
Use Cases	Scripting, REPLs, education	High-performance applications

1.2.1 Real-World Uses of Interpreters:

- **Scripting engines** in games and embedded systems
- **Domain-Specific Languages (DSLs)** for automation
- **Education and prototyping** (REPL environments)
- **Configuration languages** (e.g., TOML, JSON interpreters)

1.2.2 Common Interpreted Languages:

- **Python**: General-purpose scripting language
- **Lua**: Lightweight embeddable interpreter used in games
- **JavaScript**: Interpreted in browsers (via engines like V8)
- **Bash**: Shell scripting interpreter

1.3 Why Build Your Own Interpreter?

Creating a simple interpreter from scratch:

- **Demystifies how languages work**
- Teaches **language design**, **parsing**, **evaluation**, and **error handling**
- Improves your understanding of how Python, Lua, or similar languages behave internally
- Strengthens your **Modern C++** skills through practical application

This booklet will walk you through building an interpreter that supports:

- Arithmetic and variable assignments
- Conditionals like `if`
- Loops like `while`
- A simple REPL
- A growing grammar

1.4 Anatomy of an Interpreter

At a high level, an interpreter is composed of **three major phases**:

```
Source Code
   ↓
[ Lexical Analysis ] → Tokens
   ↓
[ Parsing ] → AST (Abstract Syntax Tree)
   ↓
[ Evaluation ] → Result / Effect
```

1.4.1 Lexical Analysis (Lexer)

Breaks the raw source into a series of **tokens** (atomic symbols) like `Number`, `Identifier`, `Operator`, `Keyword`.

1.4.2 Parsing

Takes the tokens and builds an **AST** (Abstract Syntax Tree) representing the structure and logic of the code.

1.4.3 Evaluation

Walks the AST and performs the operations described (e.g., evaluates expressions, updates variables, executes conditions and loops).

1.5 What Will You Build?

In this guide, you will build an interpreter with the following **minimal language features**:

1.5.1 Supported Features:

Table 5-2: Language Feature Examples

Feature	Example Code
Numbers	`42`
Variables	`x = 3 + 2`
Print	`print(x)`
Arithmetic	`2 + 3 * (4 - 1)`
If	`if x > 5 { print(x) }`
While	`while x < 10 { x = x + 1 }`

These form the foundation of a simple, Python-like interpreted language.

1.6 Why Use C++ to Build an Interpreter?

Historically, interpreter design has often been taught in dynamic languages. However, **Modern C++ (C++20/23)** introduces several powerful features that make C++ ideal for interpreter construction:

1.6.1 Modern C++ Advantages:

Table 6-3: Modern C++ Features in Interpreter Design

Feature	Use in Interpreter
`std::variant`	Store and dispatch different AST node types
`std::optional`	Represent failed parsing or missing values
Ranges & `string_view`	Efficient source handling in the lexer
`unique_ptr`	Safe memory ownership of tree nodes
Concepts (C++20)	Safer, constrained templates
`std::format` (C++20)	Clean and readable error formatting

Example: AST with `std::variant`

```
using Expr = std::variant<Number, Variable, BinaryOp>;

struct BinaryOp {
    std::string op;
    std::unique\_ptr<Expr> left, right;
};
```

This makes your code **type-safe**, **clean**, and **extensible**.

1.7 REPL: Read-Eval-Print Loop

Your interpreter will include a **REPL**—a loop that reads input from the user, evaluates it, and prints the result. REPLs are used in Python, Lua, Scheme, and JavaScript consoles.

1.7.1 REPL Flow:

```
User Input → Lexer → Parser → AST → Evaluator → Output
```

1.7.2 Example Session:

```
>>> x = 10
>>> x + 5
15
>>> print(x)
10
```

1.8 What You'll Learn

By the end of this booklet, you will understand:

- How source code becomes tokens
- How grammars are parsed into structured trees
- How ASTs are evaluated with runtime environments
- How to structure and test a language project
- How to apply Modern C++ features practically

1.9 Who This Guide Is For

- **Beginner-to-intermediate C++ programmers** who want to build something practical
- **Students** in compiler or programming language courses
- **Python, Lua, or JavaScript users** curious about language internals
- **Educators or library developers** building DSLs or interpreters for configuration

1.10 Summary

This chapter gave you an overview of what interpreters are, how they differ from compilers, what phases they include, and why building one using Modern C++ is both practical and enlightening.

Chapter 2

Project Setup

2.1 Introduction

Before writing any interpreter logic, we need to set up the project structure and build system. This chapter will walk you through:

- Organizing source files
- Setting up a Modern C++ toolchain using CMake
- Writing and building your first test program
- Verifying your environment with a working REPL shell entry point

This setup will scale cleanly as we add the lexer, parser, AST, and evaluator in later chapters.

2.2 Directory and File Structure

We'll use a simple, modular structure that mirrors the interpreter's architecture:

```
interpreter/
 CMakeLists.txt
 main.cpp
 lexer.hpp / lexer.cpp
```

```
parser.hpp / parser.cpp
ast.hpp
interpreter.hpp / interpreter.cpp
repl.cpp
tokens.hpp
utils.hpp
```

Each file serves a clear role:

Table 2-1: Source File Responsibilities in Interpreter Project

File	Responsibility
`main.cpp`	Entry point, REPL loop
`lexer.*`	Lexical analysis (tokenizing input)
`parser.*`	Parsing token stream into AST
`ast.hpp`	AST data structures (expression and statement nodes)
`interpreter.*`	Evaluator/Executor of AST
`repl.cpp`	Implements the Read-Eval-Print Loop
`tokens.hpp`	Defines token types and token structure
`utils.hpp`	Optional: helpers for formatting, error handling

2.3 Installing Required Tools

Make sure your system includes:

2.3.1 A Modern C++ Compiler

Compiler	Minimum Version for C++20/23 Support
GCC	10.3+ (recommended 11.0+)
Clang	12+

Compiler	Minimum Version for C++20/23 Support
MSVC	2019 (v16.9) or later

2.3.2 CMake

CMake version **3.20 or later** is recommended.
Install via package manager:

- **Linux**:

```
sudo apt install cmake g++
```

- **Windows**:
  - Install via Visual Studio Installer or cmake.org
- **macOS**:

```
brew install cmake
```

2.4 Writing `CMakeLists.txt`

Create a new file named `CMakeLists.txt` at the root:

```
cmake_minimum_required(VERSION 3.20)

project(SimpleInterpreter LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

add_executable(interpreter
    main.cpp
    repl.cpp
```

```
    lexer.cpp
    parser.cpp
    interpreter.cpp
)
```

2.4.1 Notes:

- `CMAKE_EXPORT_COMPILE_COMMANDS` is helpful for IDE integration and tooling (e.g. clangd).
- You can later split this into multiple targets for testing, modules, or libraries.

2.5 Hello World Test

Let's write a small test in `main.cpp`:

```cpp
// main.cpp
#include <iostream>

int main() {
    std::cout << "Simple Interpreter Ready" << std::endl;
    return 0;
}
```

Build and run:

```
mkdir build
cd build
cmake ..
cmake --build .
./interpreter
```

You should see:

```
Simple Interpreter Ready
```

This confirms that CMake, your compiler, and the project structure are functioning properly.

2.6 REPL Entry Point

Update `main.cpp` to prepare for REPL:

```
#include "repl.hpp"

int main() {
    start_repl();
    return 0;
}
```

Now create a basic `repl.cpp` and `repl.hpp`:

```
// repl.hpp
#pragma once
void start_repl();
cppCopyEdit// repl.cpp
#include <iostream>
#include <string>
#include "repl.hpp"

void start_repl() {
    std::string line;
    std::cout << ">>> ";
    while (std::getline(std::cin, line)) {
        if (line == "exit") break;
        std::cout << "You typed: " << line << "\n>>> ";
    }
}
```

Rebuild and run:

```
./interpreter
```

Expected behavior:

```
>>> print(5 + 2)
You typed: print(5 + 2)
>>> exit
```

This REPL will later be connected to your lexer, parser, and evaluator. For now, it serves as a sandbox for interactive input.

2.7 Optional: IDE Integration

2.7.1 Visual Studio Code:

Install extensions:

- CMake Tools
- C++ IntelliSense
- clangd or MSVC backend

Set `CMakeLists.txt` as the project root, and configure your C++ standard in `.vscode/settings.json`:

```
{
    "cmake.generator": "Ninja",
    "cmake.buildDirectory": "build",
    "C_Cpp.default.cppStandard": "c++23"
}
```

2.7.2 CLion (JetBrains):

CLion detects `CMakeLists.txt` automatically. Set C++23 in `CMake Settings`.

2.8 Summary

You've now prepared:

- A modular source directory
- A modern CMake build
- A REPL entry point

- Your toolchain and IDE (optionally)

This foundation will support all future interpreter components: lexer, parser, AST, evaluator, control flow, and more.

2.9 Exercises

1. **Modularize Your Headers**
   Move all function declarations to `.hpp` files. Use `#pragma once`.

2. **Add a Logging Utility**
   Write a simple `log(std::string)` function that can be enabled or disabled globally.

3. **Add Version Info**
   Define a `VERSION` constant in `main.cpp` or a separate config file and print it on REPL startup.

4. **Build Modes**
   Extend your `CMakeLists.txt` to support Debug/Release builds using:

   ```
   cmake -DCMAKE_BUILD_TYPE=Release ..
   ```

Chapter 3

Lexer (Tokenizer)

3.1 What Is Lexical Analysis?

Lexical analysis is the first phase of interpretation. It reads the raw source code (a plain string) and converts it into a **stream of tokens**. Tokens are atomic units of meaning—such as numbers, identifiers, operators, or keywords—that the parser will later process.

Example Input:

```
x = 3 + 4
```

Expected Tokens:

```
Token{Identifier, "x"}
Token{Assign, "="}
Token{Number, "3"}
Token{Operator, "+"}
Token{Number, "4"}
```

3.2 Token Type Definition

We begin by defining all possible token types that our language can recognize:

```
// token.hpp
enum class TokenType {
    Number,
    Identifier,
    Operator,
    Assign,
    KeywordPrint,
    KeywordIf,
    KeywordWhile,
    LeftParen,
    RightParen,
    LeftBrace,
    RightBrace,
    Semicolon,
    EndOfFile,
    Unknown
};

struct Token {
    TokenType type;
    std::string\_view lexeme;
    int line;
    int column;
};
```

3.2.1 Notes:

- We use `std::string_view` to avoid copying substrings during lexing.
- Line and column tracking helps with error reporting later.

3.3 Designing the Lexer Class

We define a class that takes the source code and produces a list of tokens.

```
// lexer.hpp
class Lexer {
public:
    Lexer(std::string\_view input);
```

```
    std::vector<Token> tokenize();

private:
    std::string\_view source;
    size_t start = 0;
    size_t current = 0;
    int line = 1;
    int column = 1;

    char peek() const;
    char advance();
    bool is\_at\_end() const;

    void skip\_whitespace();
    Token read\_number();
    Token read\_identifier\_or\_keyword();
    Token read\_operator\_or\_symbol();
};
```

3.4 Lexer Implementation

3.4.1 Constructor and Helpers

```
// lexer.cpp
Lexer::Lexer(std::string\_view input) : source(input) {}

char Lexer::peek() const {
    return is\_at\_end() ? '\0' : source[current];
}

char Lexer::advance() {
    char c = peek();
    ++current;
    ++column;
    return c;
}

bool Lexer::is\_at\_end() const {
```

```
    return current >= source.size();
}
```

3.4.2 Skipping Whitespace and Newlines

```
void Lexer::skip\_whitespace() {
    while (!is\_at\_end()) {
        char c = peek();
        if (c == ' ' || c == '\t') {
            advance();
        } else if (c == '\n') {
            ++line;
            column = 1;
            advance();
        } else {
            break;
        }
    }
}
```

3.4.3 Reading Numbers

```
Token Lexer::read\_number() {
    size\_t number\_start = current;
    while (isdigit(peek())) advance();
    auto lexeme = source.substr(number\_start, current - number\_start);
    return Token{TokenType::Number, lexeme, line, column};
}
```

3.4.4 Reading Identifiers and Keywords

```
Token Lexer::read\_identifier\_or\_keyword() {
    size\_t id\_start = current;
    while (isalnum(peek()) || peek() == '_') advance();
```

```
    auto text = source.substr(id\_start, current - id_start);

    if (text == "print") return Token{TokenType::KeywordPrint, text, line, column};
    if (text == "if")    return Token{TokenType::KeywordIf, text, line, column};
    if (text == "while") return Token{TokenType::KeywordWhile, text, line, column};

    return Token{TokenType::Identifier, text, line, column};
}
```

3.4.5 Reading Symbols and Operators

```
Token Lexer::read\_operator\_or\_symbol() {
    char c = advance();
    switch (c) {
        case '+': case '-': case '*': case '/':
            return Token{TokenType::Operator, source.substr(current - 1, 1), line, column};
        case '=':
            return Token{TokenType::Assign, source.substr(current - 1, 1), line, column};
        case '(': return Token{TokenType::LeftParen, "(", line, column};
        case ')': return Token{TokenType::RightParen, ")", line, column};
        case '{': return Token{TokenType::LeftBrace, "{", line, column};
        case '}': return Token{TokenType::RightBrace, "}", line, column};
        case ';': return Token{TokenType::Semicolon, ";", line, column};
        default:
            return Token{TokenType::Unknown, source.substr(current - 1, 1), line, column};
    }
}
```

3.5 The Tokenize Loop

```
std::vector<Token> Lexer::tokenize() {
    std::vector<Token> tokens;
    while (!is\_at\_end()) {
        skip\_whitespace();
        char c = peek();
```

```
        if (isdigit(c)) {
            tokens.push\_back(read_number());
        } else if (isalpha(c)) {
            tokens.push\_back(read_identifier\_or\_keyword());
        } else {
            tokens.push\_back(read_operator\_or\_symbol());
        }
    }

    tokens.push\_back(Token{TokenType::EndOfFile, "", line, column});
    return tokens;
}
```

3.6 Example Output

Given this input:

```
print x = 5 + 10
```

The output token stream would be:

```
[KeywordPrint, "print"]
[Identifier, "x"]
[Assign, "="]
[Number, "5"]
[Operator, "+"]
[Number, "10"]
[EndOfFile, ""]
```

3.7 Debug Utility: Print Tokens

```
void print\_tokens(const std::vector<Token>& tokens) {
    for (const auto& token : tokens) {
        std::cout << "Token{" << static\_cast<int>(token.type) << ", \""
                  << token.lexeme << "\", line " << token.line
```

```
                  << ", column " << token.column << "}\n";
    }
}
```

Use this in your REPL to debug lexer output.

3.8 Exercises

1. **Add support for comments.**
   Example: ignore anything after `//` until the end of the line.

2. **Add string literals.**
   Strings should be enclosed in quotes: `"hello world"`.

3. **Add multi-character operators.**
   Support `==`, `!=`, `<=`, `>=` in `read_operator_or_symbol()`.

4. **Track source ranges.**
   Extend the `Token` struct to include `start_pos` and `end_pos` for advanced diagnostics.

Chapter 4

Writing the Parser

4.1 What Is a Parser?

A parser is the second major phase in the interpreter. It reads the **stream of tokens** generated by the lexer and produces a **structured tree** representing the logic of the code.
This tree is called an **Abstract Syntax Tree (AST)**. Each node in the AST represents a meaningful operation: a number, a variable, an assignment, a binary operation like addition, or a statement like `print()` or `if`.

4.1.1 Input vs Output

Component	Input	Output
Lexer	Raw source code	List of `Token`
Parser	List of `Token`	AST (Abstract Syntax Tree)
Evaluator	AST	Result / Effect

4.1.2 Why Recursive Descent?

We use **recursive descent parsing** because:

- It is simple to implement.
- It works well for small, LL(1)-style grammars.
- It gives readable, clean, hand-written code.

Modern C++ makes it even more elegant with smart pointers, variants, and clear control structures.

4.2 Basic Grammar of the Language

Here is the simplified grammar for expressions and statements:

```
program     → statement* EOF

statement   → print\_stmt | assign\_stmt | if\_stmt | while\_stmt | expr\_stmt

print\_stmt  → "print" expression
assign\_stmt → IDENTIFIER "=" expression
if\_stmt     → "if" expression block
while\_stmt  → "while" expression block
expr\_stmt   → expression

block       → "{" statement* "}"

expression  → term (("+" | "-") term)*
term        → factor (("*" | "/") factor)*
factor      → NUMBER | IDENTIFIER | "(" expression ")"
```

4.3 AST Node Recap

We assume you've already defined basic AST node structures like:

```
// ast.hpp
struct Number     { double value; };
struct Variable   { std::string name; };
struct BinaryOp   { std::string op; std::unique\_ptr<Expr> left, right; };
struct Assignment { std::string name; std::unique\_ptr<Expr> expr; };
struct Print      { std::unique\_ptr<Expr> expr; };
```

```
struct Block      { std::vector<Stmt> statements; };
struct If         { std::unique\_ptr<Expr> condition; Block body; };
struct While      { std::unique\_ptr<Expr> condition; Block body; };
```

Where:

```
using Expr = std::variant<Number, Variable, BinaryOp, Assignment>;
using Stmt = std::variant<Print, If, While, Block, Assignment>;
```

4.4 The Parser Class

```
// parser.hpp
class Parser {
public:
    Parser(std::vector<Token> tokens);
    std::vector<Stmt> parse();

private:
    const std::vector<Token>& tokens;
    size\_t current = 0;

    Token peek() const;
    Token advance();
    bool match(TokenType type);
    bool check(TokenType type) const;
    bool is\_at\_end() const;

    // Parsing entry points
    Stmt parse\_statement();
    Expr parse\_expression();
    Expr parse\_term();
    Expr parse\_factor();
};
```

4.5 Helper Methods

```
Token Parser::peek() const {
    return tokens[current];
}

bool Parser::is_at_end() const {
    return peek().type == TokenType::EndOfFile;
}

Token Parser::advance() {
    if (!is_at_end()) ++current;
    return tokens[current - 1];
}

bool Parser::check(TokenType type) const {
    return !is_at_end() && peek().type == type;
}

bool Parser::match(TokenType type) {
    if (check(type)) {
        advance();
        return true;
    }
    return false;
}
```

4.6 Parsing Expressions

```
Expr Parser::parse_expression() {
    Expr expr = parse_term();
    while (match(TokenType::Operator) && (peek().lexeme == "+" || peek().lexeme == "-")) {
        std::string op = tokens[current - 1].lexeme;
        Expr right = parse_term();
        expr = BinaryOp{op, std::make_unique<Expr>(std::move(expr)),
        ↪  std::make_unique<Expr>(std::move(right))};
    }
    return expr;
```

```
}
cppCopyEditExpr Parser::parse_term() {
    Expr expr = parse_factor();
    while (match(TokenType::Operator) && (peek().lexeme == "*" || peek().lexeme == "/")) {
        std::string op = tokens[current - 1].lexeme;
        Expr right = parse_factor();
        expr = BinaryOp{op, std::make_unique<Expr>(std::move(expr)),
        ↪  std::make_unique<Expr>(std::move(right))};
    }
    return expr;
}
cppCopyEditExpr Parser::parse_factor() {
    if (match(TokenType::Number)) {
        double val = std::stod(tokens[current - 1].lexeme.data());
        return Number{val};
    }

    if (match(TokenType::Identifier)) {
        return Variable{std::string(tokens[current - 1].lexeme)};
    }

    if (match(TokenType::LeftParen)) {
        Expr expr = parse_expression();
        if (!match(TokenType::RightParen)) throw std::runtime_error("Expected ')'");
        return expr;
    }

    throw std::runtime_error("Unexpected token in factor");
}
```

4.7 Parsing Statements

```
Stmt Parser::parse_statement() {
    if (match(TokenType::KeywordPrint)) {
        Expr value = parse_expression();
        return Print{std::make_unique<Expr>(std::move(value))};
    }
```

```
    if (check(TokenType::Identifier) && tokens[current + 1].type == TokenType::Assign) {
        std::string name = std::string(advance().lexeme); // identifier
        advance(); // consume '='
        Expr value = parse_expression();
        return Assignment{name, std::make_unique<Expr>(std::move(value))};
    }

    if (match(TokenType::KeywordIf)) {
        Expr cond = parse_expression();
        Block body = parse_block();
        return If{std::make_unique<Expr>(std::move(cond)), body};
    }

    if (match(TokenType::KeywordWhile)) {
        Expr cond = parse_expression();
        Block body = parse_block();
        return While{std::make_unique<Expr>(std::move(cond)), body};
    }

    Expr expr = parse_expression();
    return Assignment{"_", std::make_unique<Expr>(std::move(expr))}; // anonymous expression
}
```

4.8 Parsing Code Blocks

```
Block Parser::parse_block() {
    if (!match(TokenType::LeftBrace)) {
        throw std::runtime_error("Expected '{' to start block");
    }

    std::vector<Stmt> stmts;
    while (check(TokenType::RightBrace) && is_at_end()) {
        stmts.push_back(parse_statement());
    }

    if (!match(TokenType::RightBrace)) {
        throw std::runtime_error("Expected '}' to close block");
    }
```

```
    return Block{std::move(stmts)};
}
```

4.9 Parsing the Program

```
std::vector<Stmt> Parser::parse() {
    std::vector<Stmt> stmts;
    while (!is_at_end()) {
        stmts.push_back(parse_statement());
    }
    return stmts;
}
```

4.10 Example: From Code to AST

Input code:

```
x = 3 + 4
print(x)
```

AST (simplified):

```
Assignment("x",
    BinaryOp("+",
        Number(3),
        Number(4)
    )
)
Print(
    Variable("x")
)
```

4.11 Error Recovery Tips

- Always check token bounds: `tokens[current + 1]` can crash.
- Track line/column in exceptions.
- Wrap parse calls in try/catch in REPL mode.

4.12 Exercises

1. **Support parentheses inside `if` and `while`**:

```
if (x > 10) { ... }
```

2. **Add comparison operators**: `==`, `!=`, <, >, `<=`, `>=`
3. **Support multi-statement blocks without braces** (optional, like Python)
4. **Implement `return` keyword** (future function support)

4.13 Summary

You've now written a parser that:

- Converts a flat token stream into a structured tree
- Understands both expressions and statements
- Supports nested arithmetic, conditionals, and loops

Chapter 5

AST Node Definitions and Tree Design

In this chapter, we define the structures needed to represent programs in memory after parsing. This phase is essential for evaluation, as it translates the parser's output into objects the evaluator can easily process.
AST stands for **Abstract Syntax Tree**, which is a hierarchical, structured representation of code. Each node in the AST corresponds to a construct in the language, such as numbers, variables, binary operations, assignments, print statements, and control flows.

5.1 Goals of This Chapter

- Define the main AST nodes for expressions and statements.
- Decide between using `std::variant` vs class inheritance.
- Implement smart memory management using `std::unique_ptr`.
- Prepare for efficient and clean tree traversal in the evaluator.

5.2 AST Nodes for Expressions

Here are the common types of expressions in a simple language:

- **Number**: Constant value like `42`
- **Variable**: Symbol reference like `x`
- **BinaryOp**: Operation like `a + b`
- **Assignment**: `x = 10`

5.2.1 Expression Node Definitions

```
// ast.hpp

struct Number {
    double value;
};

struct Variable {
    std::string name;
};

struct BinaryOp {
    std::string op;
    std::unique_ptr<Expr> left;
    std::unique_ptr<Expr> right;
};

struct Assignment {
    std::string name;
    std::unique_ptr<Expr> value;
};
```

> We wrap child expressions in `std::unique_ptr` to express ownership and enable dynamic, recursive structures.

5.3 AST Nodes for Statements

Statements include actions like:

- **Print**: `print(x)`
- **If**: conditional execution
- **While**: looping
- **Block**: group of statements

```
struct Print {
    std::unique_ptr<Expr> expr;
};

struct If {
    std::unique_ptr<Expr> condition;
    std::vector<Stmt> body;
};

struct While {
    std::unique_ptr<Expr> condition;
    std::vector<Stmt> body;
};

struct Block {
    std::vector<Stmt> statements;
};
```

5.4 Wrapping All Node Types with Variants

We now wrap the node types in type-safe discriminated unions using `std::variant`.

5.4.1 Expression Variant

```
using Expr = std::variant<Number, Variable, BinaryOp, Assignment>;
```

5.4.2 Statement Variant

```
using Stmt = std::variant<Print, If, While, Block, Assignment>;
```

This enables us to use `std::visit()` later when evaluating expressions or statements based on their actual type.

5.5 Why Use `std::variant` Instead of Inheritance?

While you could define a common base class and use virtual methods, `std::variant`:

- Is **compile-time type-safe**: no dynamic casting.
- Enables **pattern matching** with `std::visit`.
- Avoids **heap allocations** for every node type.
- Is simpler and clearer in C++20/C++23 projects.

5.5.1 Sample Use of `std::visit`

```
std::visit(overloaded {
    [](const Number& n) { return n.value; },
    [](const Variable& v) { return lookup(v.name); },
    [](const BinaryOp& b) { return evaluate_binary(b); },
    [](const Assignment& a) { return assign(a); }
}, expr);
```

5.6 Smart Memory Management with `std::unique_ptr`

In recursive structures like ASTs, we use `std::unique_ptr` to:

- Represent **ownership** of children (e.g., left/right expressions).

- Avoid manual `new`/`delete`.
- Enable **safe recursion** without memory leaks.

5.6.1 Example:

```
struct BinaryOp {
    std::string op;
    std::unique_ptr<Expr> left;
    std::unique_ptr<Expr> right;
};
```

C++20 and C++23 make working with smart pointers easier with:

- Implicit move constructors
- Structured bindings
- Clear code formatting

5.7 Example: AST for `x = 1 + 2`

Here's how the AST might look in memory:

```
Assignment{
    name = "x",
    value = std::make_unique<Expr>(
        BinaryOp{
            op = "+",
            left = std::make_unique<Expr>(Number{1}),
            right = std::make_unique<Expr>(Number{2})
        }
    )
}
```

And visually:

```
Assignment(x)
  BinaryOp(+)
      Number(1)
      Number(2)
```

This is the core data structure that the **evaluator** will walk to compute values or perform side effects like printing.

5.8 Extending the AST in the Future

Your AST can grow to support:

- **Function definitions and calls**
- **String literals**
- **Boolean logic (`&&`, `||`, `!`)**
- **Return and break statements**
- **Error tracking and debugging metadata**

Always define each new construct as a new `struct` and wrap it into the `Expr` or `Stmt` variant.

5.9 Suggested Enhancements

- Add a base interface `Node` if you want runtime polymorphism.
- Store token location in each AST node to enable detailed error reporting.
- Use `std::shared_ptr` if multiple parts of the tree need access to the same object (not typical for basic interpreters).
- Use C++23 `std::expected` to enhance construction of ASTs with error messages.

5.10 Summary

In this chapter, you:

- Defined core AST node types (Number, Variable, BinaryOp, etc.)
- Used `std::variant` to group expression and statement variants.
- Applied `std::unique_ptr` to manage memory safely.
- Prepared the AST structure for evaluation in the next stage.

Chapter 6

Evaluator – Walking the AST to Run Code

This chapter implements the core logic of your interpreter: executing the abstract syntax tree (AST) you've built from source code.
Evaluation involves traversing the AST nodes, performing operations such as arithmetic, assignments, and control flow. This process turns static structure into dynamic behavior.

6.1 What Is Evaluation?

The **evaluator** (or interpreter core) is responsible for:

- Traversing expression and statement nodes.
- Calculating numeric values.
- Managing variable values.
- Executing side effects (`print()`).
- Performing branching (`if`) and loops (`while`).

6.2 Evaluation Strategy

We'll evaluate AST nodes using:

- `std::variant` + `std::visit` for dispatching behavior based on node type.
- Recursive traversal for nested operations.
- A **symbol table** (`std::unordered_map`) for storing variable values.

6.3 Symbol Table – Storing Variables

Create a global environment:

```
#include <unordered_map>
std::unordered\_map<std::string, double> globals;
```

This stores variable names and their current values.

6.4 Evaluate Expressions

We'll write a function `evaluate_expr` that takes an `Expr` and returns a value:

```
double evaluate_expr(const Expr& expr);
```

6.4.1 Expression Cases

```
double evaluate_expr(const Expr& expr) {
    return std::visit(overloaded{
        [](const Number& n) {
            return n.value;
        },
        [](const Variable& v) {
            return globals.at(v.name); // Throws if not found
        },
        [](const BinaryOp& b) {
```

```
            double left = evaluate_expr(*b.left);
            double right = evaluate_expr(*b.right);
            if (b.op == "+") return left + right;
            if (b.op == "-") return left - right;
            if (b.op == "*") return left * right;
            if (b.op == "/") return left / right;
            throw std::runtime_error("Unknown operator: " + b.op);
        },
        [](const Assignment& a) {
            double val = evaluate_expr(*a.value);
            globals[a.name] = val;
            return val;
        }
    }, expr);
}
```

This uses `std::visit` to pattern-match the type inside `std::variant`.

6.5 Evaluate Statements

Statements don't return values, they perform actions. Define a function:

```
void execute_stmt(const Stmt& stmt);
```

6.5.1 Statement Cases

Conditionals are treated as "true" if not zero.

6.6 Example: Evaluating a Full Program

For the input:

```
x = 3 + 4;
print(x);
```

The evaluation does:

- Compute `3 + 4 → 7`
- Store `x = 7` in `globals`
- Output `7` to the console

6.7 Tips for Cleaner Code

- Use an `Environment` class to manage scopes in future chapters.
- Add tracing/logging for debugging evaluation steps.
- Use exceptions to handle undefined variables or bad operations.
- For complex languages, separate **ExprEvaluator** and **StmtExecutor**.

6.8 Advanced: Boolean Expressions

You can later extend `evaluate_expr` to support:

- `==`, `!=`, `<`, `>`, `<=`, `>=`
- Logical operators: `&&`, `||`

Return `1.0` for `true`, `0.0` for `false`, or introduce a `Value` type using `std::variant<double, bool, std::string>`.

6.9 Summary

In this chapter, you've implemented the **interpreter core**:

- Recursively evaluated expression trees.
- Stored and retrieved variables.
- Performed side effects like `print()`.
- Executed control flow (`if`, `while`, `block`).
- Used modern C++ features (`std::variant`, `std::visit`, smart pointers).

Chapter 7

Statements and REPL – Making Your Language Interactive

Now that your interpreter can parse and evaluate programs, it's time to make it interactive with a **REPL**: a Read–Eval–Print Loop. This allows users to enter code line by line and see immediate results, just like in Python.

7.1 What Is a REPL?

REPL stands for:

- **Read** – Accept user input (a line of source code).
- **Eval** – Tokenize, parse, and evaluate it.
- **Print** – Output the result or any errors.
- **Loop** – Repeat the cycle until the user exits.

REPLs are great for:

- Debugging
- Prototyping

- Learning
- Interactive environments

7.2 Add Support for `print(expr)`

First, extend your grammar and parser to recognize `print()` calls.

7.2.1 Parser Support

Add a new AST node:

```
struct Print {
    std::unique\_ptr<Expr> expr;
};
```

Add a variant entry:

```
using Stmt = std::variant<Print, Assignment, Block, If, While>;
```

Update the parser:

```
Stmt parse_statement() {
    if (match("print")) {
        consume("(");
        auto expr = parse_expression();
        consume(")");
        return Print{ std::move(expr) };
    }
    // other cases...
}
```

7.2.2 Evaluator Support

Update `execute_stmt`:

```
[](const Print& p) {
    std::cout << evaluate_expr(*p.expr) << std::endl;
},
```

Now your language supports visible output.

7.3 Building the REPL Loop

Write a loop in `repl.cpp`:

```
void repl() {
    while (true) {
        std::cout << ">> ";
        std::string line;
        if (!std::getline(std::cin, line)) break;
        if (line.empty()) continue;

        try {
            auto tokens = tokenize(line);
            auto stmt = parse_statement(tokens);
            execute_stmt(stmt);
        } catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << '\n';
        }
    }
}
```

Add `repl();` to `main()`.

7.4 Example REPL Session

```
>> x = 5 + 3
>> print(x)
8
>> y = x * 2
>> print(y)
16
```

Every line is processed independently. State (variables) is preserved across lines.

7.5 Enhancements and Features

You can improve the REPL with:

- **Multi-line input**: detect { or (to allow block or multi-line entry.
- **Command shortcuts**: support `:exit`, `:vars`, `:help`.
- **History**: store previous commands (can use external libraries).
- **Autocomplete**: suggest variable names or keywords (advanced).

7.6 REPL and Real-Time Feedback

Why use REPL in a language?

- Instant feedback accelerates learning.
- Encourages experimentation.
- Enables scripting tools and dynamic control.
- Makes the language more interactive and fun to use.

REPL is also useful during development to test language features quickly.

7.7 Organizing REPL in Codebase

Split your REPL logic into a separate file `repl.cpp`:

```
void start_repl() {
    // same code from above
}
```

Declare in `repl.hpp`:

```
void start\_repl();
```

Update `main.cpp`:

```
int main() {
    std::cout << "Simple Interpreter v0.1\n";
    start\_repl();
    return 0;
}
```

7.8 Summary

You now have:

- `print()` support to output values.
- A full interactive loop that reads and executes code.
- A cleaner way for users to interact with your interpreter.

Chapter 8

Control Flow – If and While Statements

So far, your language supports variables, expressions, and output through `print()`. To make it truly useful, you now need **control flow constructs**—statements that allow conditional execution (`if`) and repetition (`while`).
This chapter explains how to parse and evaluate both.

8.1 Adding the `if` Statement

8.1.1 Syntax

```
if (condition) {
    statements...
}
```

Optionally:

```
if (condition) {
    statements...
} else {
```

```
    other_statements...
}
```

8.1.2 AST Node

```
struct If {
    std::unique_ptr<Expr> condition;
    std::vector<Stmt> then_branch;
    std::vector<Stmt> else_branch;
};
```

8.1.3 Parser Support

Extend your `parse_statement()` function:

```
Stmt parse_if() {
    consume("if");
    consume("(");
    auto condition = parse_expression();
    consume(")");
    auto then_branch = parse_block();

    std::vector<Stmt> else_branch;
    if (match("else")) {
        else_branch = parse_block();
    }

    return If{ std::move(condition), std::move(then_branch), std::move(else_branch) };
}
```

8.2 Adding the `while` Loop

8.2.1 Syntax

```
while (condition) {
    statements...
}
```

8.2.2 AST Node

```
struct While {
    std::unique_ptr<Expr> condition;
    std::vector<Stmt> body;
};
```

8.2.3 Parser Support

```
Stmt parse_while() {
    consume("while");
    consume("(");
    auto condition = parse_expression();
    consume(")");
    auto body = parse\_block();
    return While{ std::move(condition), std::move(body) };
}
```

Update `parse_statement()` to dispatch based on keyword (`if`, `while`, `print`, etc.).

8.3 Parsing Blocks

8.3.1 Code Block Syntax

Blocks are delimited by { and }:

```
{
    statement1;
```

```
    statement2;
}
```

8.3.2 Parser Utility

```
std::vector<Stmt> parse\_block() {
    consume("{");
    std::vector<Stmt> statements;
    while (!match("}")) {
        statements.push_back(parse\_statement());
    }
    return statements;
}
```

8.4 Evaluator Support

Extend your `execute_stmt()` dispatch logic:

```
std::visit(overloaded {
    ...
    [&](const If& stmt) {
        if (evaluate_expr(*stmt.condition)) {
            for (const auto& s : stmt.then\_branch) execute\_stmt(s);
        } else {
            for (const auto& s : stmt.else\_branch) execute\_stmt(s);
        }
    },
    [&](const While& stmt) {
        while (evaluate_expr(*stmt.condition)) {
            for (const auto& s : stmt.body) execute\_stmt(s);
        }
    }
}, stmt);
```

8.5 Simple Example

```
>> x = 5
>> if (x > 3) { print(100) }
100
>> y = 0
>> while (y < 3) { print(y); y = y + 1 }
0
1
2
```

8.6 Representing Code Blocks

In your AST and parser:

- Code blocks are `std::vector<Stmt>`
- They can be nested inside `If`, `While`, and `Block`
- Statements are executed in order

This is essential for structured control flow.

8.7 Scoping and Variable Isolation

You can implement block scope using a **scope stack**:

```
std::vector<std::unordered_map<std::string, double>> scopes;
```

On block entry:

```
scopes.push_back({});
```

On block exit:

```
scopes.pop_back();
```

For lookups:

- Search from top down
- Update nearest match

This enables variable shadowing and better organization.

8.8 Future Extensions

Later, you can support:

- `break` and `continue`
- `for` loops
- `switch-case`
- Boolean short-circuiting

8.9 Summary

You've added:

- Full support for `if` and `while`
- Structured blocks using {} and `std::vector<Stmt>`
- Optional scoping support for block-local variables

With control flow now implemented, your interpreter becomes a true mini-language capable of branching and looping.

Chapter 9

Error Handling

Interpreters must handle invalid input gracefully. A single syntax error or runtime failure shouldn't crash the whole program. This chapter introduces clean and modern ways to catch, report, and recover from errors in your interpreter using C++20 and C++23.

9.1 Types of Errors

Errors typically fall into two categories:

1. **Syntax Errors** – Found during tokenization or parsing (e.g., unexpected symbol, unmatched parentheses).
2. **Runtime Errors** – Occur during evaluation (e.g., division by zero, undefined variable).

Handling both is critical for a reliable user experience.

9.2 Adding Line and Column Information

9.2.1 Token Metadata

Enhance your `Token` struct:

```
struct Token {
    TokenType type;
    std::string\_view lexeme;
    int line;
    int column;
};
```

Update the lexer to track line and column as characters are processed.

9.3 Syntax Error Reporting

Throw errors during parsing:

```
void error(const Token& token, const std::string& message) {
    throw std::runtime\_error("Syntax Error at line " + std::to_string(token.line) +
                             ", column " + std::to\_string(token.column) + ": " + message);
}
```

Use this in `consume()`:

```
Token consume(TokenType expected) {
    if (current().type == expected)
        return advance();
    error(current(), "Expected token type: " + token\_type\_to_string(expected));
}
```

C++20's `std::format` (if available) makes formatting even cleaner.

9.4 Runtime Error Handling

You can detect problems during evaluation, such as:

- Division by zero
- Using undeclared variables
- Invalid operator usage

```
double evaluate\_binary\_op(const std::string& op, double left, double right) {
    if (op == "/") {
        if (right == 0)
            throw std::runtime\_error("Runtime Error: Division by zero");
        return left / right;
    }
    ...
}
```

Use a top-level try/catch block in the REPL to recover:

```
try {
    auto tokens = tokenize(line);
    auto ast = parse(tokens);
    evaluate(ast);
} catch (const std::exception& ex) {
    std::cerr << ex.what() << "\n";
}
```

9.5 Optional and Result Types

C++20 provides `std::optional<T>` for expressing failure without exceptions:

```
std::optional<Token> try_consume(TokenType expected) {
    if (current().type == expected)
        return advance();
    return std::nullopt;
}
```

You can also define a simple `Result<T>` type:

```
template<typename T>
struct Result {
    T value;
    std::string error;
    bool is\_ok;
};
```

And use it instead of throwing:

```
Result<double> safe\_divide(double a, double b) {
    if (b == 0) return {0.0, "Division by zero", false};
    return {a / b, "", true};
}
```

This approach avoids exceptions, improves control, and supports future error handling improvements.

9.6 Highlighting Errors with Context

Output errors with context:

```
Error: Unexpected token '=' at line 3, column 5
--> 3 | x == 5 + ;
              ^
```

You can display the original line and use spacing to point to the column position. Store the full source lines for this purpose.

9.7 Structured Error Classes

Use error classes for better control:

```
class InterpreterError : public std::runtime\_error {
public:
    InterpreterError(const std::string& msg, int line, int column)
        : std::runtime\_error(msg), line(line), column(column) {}
    int line, column;
};
```

Or specialize for different types (e.g., `SyntaxError`, `RuntimeError`) for fine-grained catching.

9.8 REPL Integration

Wrap the whole REPL loop:

```
while (true) {
    try {
        std::string line;
        std::getline(std::cin, line);
        auto tokens = tokenize(line);
        auto ast = parse(tokens);
        evaluate(ast);
    } catch (const InterpreterError& e) {
        std::cerr << "Error at line " << e.line << ", column " << e.column << ": " << e.what() << "\n";
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}
```

This prevents the interpreter from terminating on every error.

9.9 Summary

This chapter introduced error-handling infrastructure:

- Track line/column metadata in tokens
- Report syntax and runtime errors clearly
- Catch exceptions at the REPL level
- Use `std::optional` or `Result<T>` for error-aware logic

A robust error system helps both beginners and advanced users quickly debug issues and makes the language feel more complete.

Chapter 10

Final Touches and Next Steps

In this final chapter, we summarize the features already implemented and provide a roadmap for improving and scaling your interpreter. Even though the core functionality is in place, interpreters are extensible by design—and Modern C++ makes the process cleaner, safer, and more expressive than ever before.

10.1 Recap of What's Been Built

You've successfully built a functioning interpreter in under 50 pages using C++20/C++23.
Your interpreter now supports:

- **Lexical analysis** of identifiers, numbers, operators, and control symbols.
- **Parsing** expressions, assignments, and statements using recursive descent.
- **AST creation** via modern structures (`std::variant`, `std::unique_ptr`).
- **Evaluation** of binary operations, variables, and assignments.
- **Control flow** with `if` and `while`.
- **Printing output** with `print()`.
- **A REPL (Read-Eval-Print Loop)** for line-by-line execution.

- **Basic error handling** for both syntax and runtime problems.

Technologies used:

- `std::variant` – For managing typed AST nodes.
- `std::unique_ptr` – For safe, automatic memory management.
- `std::optional` or exceptions – For error reporting.
- CMake – For cross-platform project setup.
- C++20/23 – For expressive and maintainable syntax.

This is already a minimal language engine capable of interpreting simple programs. The architecture is designed for growth.

10.2 Ideas for Extending the Language

The current interpreter is a solid foundation. Here's what you can add next:

1. **User-Defined Functions**
   - Allow defining and calling functions with parameters.
   - Maintain a call stack and local scope for each call.

   Example Syntax:

```
def add(a, b) {
    return a + b;
}
print(add(2, 3));
```

2. **String Support**
   - Add `String` to your token types and AST variants.
   - Implement string operations (concatenation, length, etc.).

Example:

```
name = "John";
print("Hello, " + name);
```

3. **File I/O Support**

   - Add `read(filename)` and `write(filename, data)` built-in functions.
   - Use `std::filesystem` and `std::ifstream/std::ofstream`.

   Example:

```
data = read("input.txt");
write("output.txt", data);
```

10.3 Suggestions for Larger-Scale Interpreters

Once your interpreter gets larger, scale it with professional design choices:

1. **Modular Design with Namespaces or C++ Modules**

   Break components into smaller libraries:

   - Lexer
   - Parser
   - AST
   - Evaluator
   - Built-ins
   - REPL

   This structure improves maintainability and compile times.

2. **Enhanced Type System**

   Support:

- Booleans
- Arrays
- Maps
- Custom types

Use `std::variant` to model more complex type hierarchies.

3. **Scope Management**

   Replace single symbol table with a scope chain or environment stack. This supports:

   - Local variables
   - Closures
   - Shadowing

4. **Function Call Stack**

   Track:

   - Call depth
   - Local variables
   - Return addresses

   A `Frame` object can hold this information.

5. **Import and Module System**

   Allow users to separate code into files and reuse components.

   Example:

```
import "math.lang"
print(add(2, 3));
```

6. **AST Optimization**

   Perform constant folding, dead code elimination, and simple inlining.

10.4 Language Design Ideas

Here are additional concepts to explore:

- **Garbage Collection**: Swap `std::unique_ptr` with shared ownership or design a GC.
- **JIT Compilation**: Use libraries like LLVM or asmjit to compile hot paths.
- **Debugging Tools**: AST printers, breakpoints, step-by-step evaluation.
- **Interactive Documentation**: Add built-in help and error hints.

10.5 Learning Path Forward

Once you've built this interpreter, you have a strong understanding of language implementation. You can now study:

- Compiler design (start building a bytecode or native compiler).
- Advanced parsing techniques (LR parsers, PEGs).
- Virtual machines and register-based evaluation.
- Language design principles (syntax, semantics, user experience).

Books to continue with:

- *Crafting Interpreters* by Bob Nystrom
- *Programming Language Pragmatics* by Michael Scott
- *Modern C++ Design* by Andrei Alexandrescu

10.6 Closing Thoughts

Building an interpreter from scratch is one of the most rewarding projects a programmer can pursue. It sharpens your skills in:

- Syntax design

- Recursive thinking
- Memory management
- C++ idioms and type systems
- Real-world architecture

With just Modern C++ and standard libraries, you've written a working language—one you can grow, enhance, and share.

Whether you plan to design a full scripting engine, teach programming, or just enjoy the craftsmanship of language building, this project is a powerful launch point.

Congratulations. You're now not only a Modern C++ developer, but also a language designer.

Appendices

Appendix A: Full Token and Tokenizer System

10.6.1 Token Types

```
enum class TokenType {
    Number,
    Identifier,
    String,
    Operator,
    Assign,
    Semicolon,
    LeftParen,
    RightParen,
    LeftBrace,
    RightBrace,
    Comma,
    KeywordPrint,
    KeywordIf,
    KeywordElse,
    KeywordWhile,
    KeywordFunc,
    KeywordReturn,
    EndOfFile,
    Unknown
};
```

10.6.2 Token Structure

```
struct Token {
    TokenType type;
    std::string_view lexeme;
    int line;
    int column;
};
```

10.6.3 Tokenizer Extensions

Add support for:

- **String literals**: Detect and handle quoted text.
- **Semicolons**: Allow statement separation.
- **Function keywords**: `func`, `return`, etc.

Example for string handling in lexer:

```
if (ch == '"') {
    ++pos;
    size_t start = pos;
    while (pos < source.size() && source[pos] != '"') ++pos;
    std::string_view str = source.substr(start, pos - start);
    tokens.push_back({ TokenType::String, str, line, col });
    ++pos; ++col;
    continue;
}
```

Appendix B: CMake Enhancements for Multi-File Projects

10.6.4 Modular CMake Structure

```
add_library(lexer lexer.cpp lexer.hpp)
add_library(parser parser.cpp parser.hpp)
add_library(ast ast.cpp ast.hpp)
add_library(interpreter interpreter.cpp interpreter.hpp)

add_executable(interpreter_cli main.cpp repl.cpp)
target\_link\_libraries(interpreter\_cli PRIVATE lexer parser ast interpreter)
```

10.6.5 Add Unit Test Target

```
enable_testing()
add_executable(test\_runner tests/test\_main.cpp)
add_test(NAME LexerTest COMMAND test_runner)
```

10.7 Appendix C: Sample Unit Tests Using `Catch2`

```
#define CATCH_CONFIG_MAIN
#include <catch2/catch.hpp>
#include "lexer.hpp"

TEST_CASE("Lex simple expression") {
    std::string src = "x = 42 + 3";
    auto tokens = tokenize(src);
    REQUIRE(tokens.size() == 5);
    REQUIRE(tokens[0].type == TokenType::Identifier);
    REQUIRE(tokens[2].type == TokenType::Number);
}
```

10.7.1 Notes:

- Use `Catch2`, `doctest`, or `Google Test`.
- Keep tests for Lexer, Parser, and Evaluator separate.
- Automate testing in CMake.

Appendix D: Template for Expression Evaluation

```
std::unordered_map<std::string, double> globals;

double eval_expr(const Expr& expr) {
    return std::visit(overloaded{
        [](const Number& n) { return n.value; },
        [](const Variable& v) -> double {
            auto it = globals.find(v.name);
            if (it != globals.end()) return it->second;
            throw std::runtime_error("Undefined variable: " + v.name);
        },
        [](const BinaryOp& b) -> double {
            double l = eval_expr(*b.left);
            double r = eval_expr(*b.right);
            if (b.op == "+") return l + r;
            if (b.op == "-") return l - r;
            if (b.op == "*") return l * r;
            if (b.op == "/") return l / r;
            throw std::runtime_error("Unknown operator");
        },
        [](const Assignment& a) -> double {
            double val = eval_expr(*a.expr);
            globals[a.name] = val;
            return val;
        }
    }, expr);
}
```

Appendix E: Common Errors and Fixes

Error Type	Description	Solution
Undefined symbol	Variable used before assignment	Check variable presence in symbol table
Token mismatch	Invalid character in input	Add default handling in tokenizer

Error Type	Description	Solution
Empty AST	Parser failed to return a root node	Ensure parser handles all valid inputs
Infinite loop	In `while`, condition never becomes false	Use debug print to inspect variable state
Missing tokens	Lexer skipped important characters	Carefully handle all punctuation cases

Appendix F: Adding a Simple Function System (Optional Extension)

10.7.2 Function Node

```
struct Function {
    std::string name;
    std::vector<std::string> params;
    std::vector<Stmt> body;
};
```

10.7.3 Call Node

```
struct Call {
    std::string funcName;
    std::vector<Expr> arguments;
};
```

10.7.4 Extend Evaluator

```
std::unordered_map<std::string, Function> functions;
```

```
Value eval_call(const Call& call) {
    auto& func = functions[call.funcName];
    // Create new environment, bind parameters to arguments
    // Evaluate body and return last value or `return` statement value
}
```

Appendix G: Advanced Grammar Sample (Extended Language)

```
program        → statement*
statement      → expr_stmt | if_stmt | while_stmt | func_decl
expr_stmt      → IDENT "=" expression ";" | "print" "(" expression ")" ";"
if_stmt        → "if" "(" expression ")" block ("else" block)?
while_stmt     → "while" "(" expression ")" block
func_decl      → "func" IDENT "(" param_list? ")" block
expression     → term (( "+" | "-" ) term)*
term           → factor (( "*" | "/" ) factor)*
factor         → NUMBER | STRING | IDENT | "(" expression ")" | call_expr
call_expr      → IDENT "(" arguments? ")"
```

Appendix H: Building and Running Instructions

10.7.5 Building with CMake

```
mkdir build
cd build
cmake ..
cmake --build .
```

10.7.6 Running the REPL

```
./interpreter_cli
```

10.7.7 Sample Input

```
x = 5
y = x + 7
print(y)
```

Appendix I: Sample REPL Loop with Error Catching

```
int main() {
    while (true) {
        std::cout << ">>> ";
        std::string input;
        if (!std::getline(std::cin, input)) break;

        try {
            auto tokens = tokenize(input);
            auto ast = parse(tokens);
            evaluate(ast);
        } catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << "\n";
        }
    }
    return 0;
}
```

Appendix J: Minimal AST Visualization for Debugging

```
void print_ast(const Expr& expr, int indent = 0) {
    std::visit(overloaded {
        [indent](const Number& n) {
            std::cout << std::string(indent, ' ') << "Number(" << n.value << ")\n";
        },
        [indent](const Variable& v) {
            std::cout << std::string(indent, ' ') << "Variable(" << v.name << ")\n";
        },
```

```
        [indent](const BinaryOp& b) {
            std::cout << std::string(indent, ' ') << "BinaryOp(" << b.op << ")\n";
            print_ast(*b.left, indent + 2);
            print_ast(*b.right, indent + 2);
        },
        [indent](const Assignment& a) {
            std::cout << std::string(indent, ' ') << "Assignment(" << a.name << ")\n";
            print_ast(*a.expr, indent + 2);
        }
    }, expr);
}
```

References

Modern C++ Standards

1. **ISO/IEC 14882:2020** – Programming Language C++ (C++20 Standard)
2. **ISO/IEC 14882:2023** – Programming Language C++ (C++23 Standard Draft and Features)
3. **cppreference.com** – https://en.cppreference.com
    - Comprehensive documentation for `std::variant`, `std::optional`, `std::visit`, `std::unique_ptr`, `std::ranges`, `std::format`, and more.

Interpreter and Compiler Construction

1. **Crafting Interpreters** – Robert Nystrom
    - While it is written in Java, the structure and methodology were adapted to C++ in this booklet.
    - Focused on recursive descent parsers and tree-walking interpreters.
2. **Let's Build a Simple Interpreter** – Ruslan Spivak
    - A minimalist Python-based tutorial that inspired the breakdown into lexer → parser → interpreter.
    - Converted and modernized into C++ for this project.
3. **The Super Tiny Compiler** – James Kyle

- A well-known JavaScript-focused compiler demo; adapted to C++ to explain abstract syntax tree construction in simple steps.

C++ Resources Focused on Modern Features

1. **Anthony Williams** – *C++ Concurrency in Action* (Second Edition, updated for C++20)
   - Relevant in memory management, smart pointers, and resource safety using RAII.
2. **Jason Turner** – *C++ Best Practices* and talks from CppCon and Meeting C++
   - Advocates for clean, modern C++ style, especially use of `std::variant` and `constexpr`.
3. **Ben Deane & Jason Turner** – *Variant Visitors in Modern C++*
   - Presenting practical use of `std::variant` and `std::visit` for expression evaluation.
4. **Herb Sutter** – C++ Language Evolution Proposals (ISO papers)
   - His contributions around pattern matching, language safety, and the future of error handling in C++23 and beyond.

CMake and Tooling

1. **CMake Documentation** – https://cmake.org/documentation
   - For build system setup, modular configuration, linking multiple source files, and toolchain usage for C++20/23.
2. **JetBrains CLion Documentation**
   - Advanced IDE support for C++20 and integration with CMake and unit testing frameworks.
3. **Catch2 Framework** – https://github.com/catchorg/Catch2
   - Used in the booklet to demonstrate unit testing of the lexer and parser.

Practical Community Resources and Examples

1. **Compiler Explorer (Godbolt)** – https://godbolt.org
   - Used to test and validate modern C++ snippets in real-time with support for latest compilers (GCC, Clang, MSVC).
2. **GitHub Projects for Educational Interpreters**:
   - Several open-source projects were referenced to cross-check idioms and best practices, including:
     - `Starlang` (C++-based)
     - `Cxx-Lox`
     - `kaleidointerpreter` (LLVM/C++)

Academic Foundations

1. **"Programming Language Pragmatics" by Michael L. Scott**
   - Provides theoretical background for grammar rules, recursive descent parsing, and runtime environments.
2. **"Modern Compiler Implementation in C" by Andrew W. Appel**
   - Though written in C, the data structures and flow served as a foundation for the parser and evaluator designs.