https://ForgeVM.org

# Quick Guide to Build Your Own Simple Interpreter

# Using Modern C++

Prepared By Ayman Alheraki

Second Edition

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

Prepared by Ayman Alheraki

ForgeVM.org

March 2026

# Contents

# Author's Preface

The design of programming languages is one of the most fascinating and intellectually rewarding areas of computer science. Many programmers, after several years of practical experience, eventually reach a moment when they begin to ask a deeper question:

*How does a programming language actually work?*

Behind every programming language lies a system that transforms human-readable instructions into executable behavior. Modern compilers and interpreters are extremely sophisticated systems built by large teams of engineers, yet the fundamental principles behind them remain surprisingly elegant and approachable when explained in a structured way.

This booklet was written with a clear purpose: to help serious learners build their **first real interpreter**.

Many tutorials available today demonstrate small "toy" interpreters that evaluate simple arithmetic expressions. While such examples are useful for introducing concepts like lexical analysis and parsing, they often stop before reaching the stage where the system resembles a **real programming language runtime**.

In this guide, we go further.

The interpreter presented here is implemented using **Modern C++ (C++20/23)**, but with a very practical philosophy: **using Modern C++ essentially as a safer and more expressive version of the C language**.

In other words, the interpreter is designed in a clear procedural style, similar to how classic systems software was written in C, while taking advantage of a few powerful Modern C++ tools that significantly improve safety and clarity.
These include:

- The **Standard Template Library (STL)** for containers such as `std::vector` and `std::unordered_map`.
- **Smart pointers** such as `std::unique_ptr` for safe ownership of recursive data structures like abstract syntax trees.
- **`std::variant`** and **`std::visit`** for representing heterogeneous AST nodes without complex inheritance hierarchies.
- Lightweight utilities such as `std::optional`, `std::string_view`, and modern standard algorithms.

By combining the clarity of classic C-style system design with selected Modern C++ facilities, we can build a language runtime that is both **simple to understand and robust to implement**.
Throughout this booklet we construct a compact yet complete interpreter that includes the essential components found in real language implementations:

- A **Lexer** that transforms raw source code into tokens.
- A **Parser** that builds a structured representation of the program.
- An **Abstract Syntax Tree (AST)** describing program structure.
- An **Interpreter Engine** capable of evaluating expressions and executing statements.

- A **Symbol Table** for managing variables and runtime state.
- Support for **control flow** such as conditional execution and loops.
- A minimal but functional **interactive scripting environment** using a REPL.

The goal is not to build a massive compiler infrastructure, but rather to design a **small and understandable interpreter that behaves like a real language runtime**. Each component is introduced step by step, with emphasis on clarity, architecture, and practical implementation.

This guide is also connected to my ongoing work on the **ForgeVM project**, which explores the broader ecosystem of programming language engineering, including assemblers, toolchains, and virtual machine design. While ForgeVM itself is a much larger system, the ideas presented here represent many of the foundational concepts that appear in real language runtimes.

In parallel with this booklet, I am also preparing a much larger work dedicated to programming language design and implementation, where these topics will be explored in significantly greater depth. However, experience has shown that learners benefit enormously from a **focused and practical starting point**. This booklet is intended to provide exactly that.

By the end of this guide, readers will not only understand how interpreters work internally, but will also possess the knowledge required to begin experimenting with their own language designs.

If this booklet inspires even a single programmer to build their own programming language, then it has achieved its goal.

`Ayman Alheraki`

# Chapter 1

# Introduction

## 1.1 What Is an Interpreter?

An interpreter is a software system that **reads source code and executes it directly at runtime**, without first translating the entire program into a standalone executable binary. Instead of producing machine code ahead of time, the interpreter analyzes the program and performs the requested operations immediately.

This execution model makes interpreters especially suitable for **dynamic**, **interactive**, and **rapid-development environments**. Many modern languages rely on interpreter-based execution models, including Python, Lua, and JavaScript.

Unlike traditional compilers that transform source code into optimized machine code before execution, interpreters operate **during program execution**, typically evaluating code **statement by statement** or **expression by expression**. This approach simplifies experimentation and interactive programming, though it may introduce additional runtime overhead.

Despite this, interpreters remain widely used because they offer flexibility, simplicity of deployment, and strong support for interactive environments.

## 1.2 Key Characteristics of Interpreters

Table 2-1: Interpreter vs Compiler Comparison

| Feature | Interpreter | Compiler |
|---|---|---|
| Execution Model | Executes program instructions directly | Produces machine code before execution |
| Runtime Speed | Typically slower due to runtime evaluation | Typically faster due to precompiled machine code |
| Error Detection | Errors appear during execution | Errors detected during compilation phase |
| Development Style | Ideal for interactive and scripting workflows | Ideal for performance-critical applications |
| Typical Usage | Scripting languages, REPL systems, DSLs | System software, games, operating systems |

### 1.2.1 Real-World Uses of Interpreters

Interpreters are used extensively across many areas of software engineering:

- **Scripting engines** embedded in applications and game engines
- **Domain-Specific Languages (DSLs)** used for automation and configuration
- **Interactive development environments** such as REPL systems
- **Configuration languages** used in modern infrastructure systems

- **Embedded scripting systems** inside large applications

Examples include scripting systems used in game engines, configuration languages for build systems, and automation tools in modern development pipelines.

### 1.2.2 Common Interpreted Languages

Some well-known languages that rely heavily on interpreter-based execution include:

- **Python** — widely used in scripting, automation, and data science
- **Lua** — a lightweight embeddable interpreter frequently used in game engines
- **JavaScript** — executed inside web browsers through engines such as V8
- **Bash** — the command interpreter used in Unix-like operating systems

Many of these languages actually use a hybrid approach combining interpretation, bytecode execution, and Just-In-Time (JIT) compilation.

## 1.3 Why Build Your Own Interpreter?

Designing and implementing an interpreter from scratch is one of the best ways to deeply understand how programming languages work internally.
Building an interpreter allows you to:

- **Understand the internal structure of programming languages**
- Learn how **source code is analyzed and executed**
- Gain practical experience with **parsing**, **language design**, and **runtime systems**

- Develop stronger architectural thinking about software systems
- Apply advanced **Modern C++ programming techniques**

Even a small interpreter demonstrates many core ideas used in real language runtimes. In this booklet, you will build an interpreter that supports:

- Arithmetic expressions
- Variable declarations and assignments
- Conditional execution using `if`
- Loop constructs such as `while`
- A simple interactive REPL
- A gradually extensible language grammar

Although the language implemented in this guide is intentionally small, the architectural ideas are similar to those used in real programming language implementations.

## 1.4 Design Philosophy: C-Style Interpreter Architecture

Many Modern C++ codebases rely heavily on object-oriented programming, class hierarchies, and virtual dispatch. While these techniques can be useful in some domains, they are not required to build a clean and capable interpreter.
In this guide we intentionally adopt a **C-style architecture implemented in Modern C++**. That means the interpreter avoids classical object-oriented design

and instead emphasizes simple data structures, explicit control flow, and procedural organization.
The implementation relies primarily on:

- Plain `struct` data structures
- Free functions instead of member methods whenever practical
- Standard Library containers such as `std::vector` and `std::unordered_map`
- Smart pointers such as `std::unique_ptr` for memory safety
- `std::variant` for representing AST nodes in a type-safe way

This style gives us several advantages:

- **A design closer to traditional compiler and interpreter implementations**
- **Simple control flow that is easy to trace and debug**
- **Less abstraction overhead and fewer indirection layers**
- **A data-oriented structure that feels natural to systems programmers**

In other words, the project uses C++ as a **better C for systems programming tasks** rather than as a purely object-oriented language. We benefit from Modern C++ features such as `std::vector`, `std::string`, `std::string_view`, `std::variant`, and `std::unique_ptr`, while still keeping the overall architecture procedural and explicit. This philosophy resembles the engineering style used in many successful systems projects where clarity, portability, and control over data layout matter more than class-heavy design.

## 1.5 Anatomy of an Interpreter

At a high level, interpreters typically follow a sequence of processing stages. The source code passes through several transformations before the program's behavior is executed.

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

Each stage is responsible for transforming the program into a representation that is easier to analyze and execute.

### 1.5.1 Lexical Analysis (Lexer)

Lexical analysis is the process of converting raw source code into a stream of **tokens**. Tokens are the smallest meaningful elements of the language.
Examples of tokens include:

- Numbers
- Identifiers
- Operators
- Keywords
- Delimiters

For example, the expression:

```
x = 3 + 2;
```

would be converted into tokens such as:
`Identifier(x)`, `Assignment`, `Number(3)`, `Plus`, `Number(2)`

### 1.5.2 Parsing

The parser takes the sequence of tokens produced by the lexer and constructs a structured representation called an **Abstract Syntax Tree (AST)**.
The AST represents the logical structure of the program rather than the exact textual form of the source code.
For example:

```
3 + 4 * 5
```

would produce an AST representing multiplication occurring before addition according to operator precedence rules.

### 1.5.3 Evaluation

The final stage walks through the AST and performs the operations described by each node.
The interpreter evaluates expressions, updates variables, executes control-flow constructs, and produces program output.
This stage is often referred to as the **runtime engine** of the interpreter.

## 1.6 What Will You Build?

In this guide, you will construct a small interpreter that supports a minimal but practical set of language features.

### 1.6.1 Supported Language Features

Table 6-2: Language Feature Examples

| Feature | Example Code |
|---|---|
| Numbers | `42` |
| Variables | `x = 3 + 2` |
| Printing | `print(x)` |
| Arithmetic Expressions | `2 + 3 * (4 - 1)` |
| Conditional Statements | `if x > 5 { print(x) }` |
| Looping Constructs | `while x < 10 { x = x + 1 }` |

These capabilities form the foundation of a small interpreted scripting language similar in spirit to Python or Lua.

## 1.7 Why Use C++ to Build an Interpreter?

Interpreter development is often demonstrated using dynamic languages for simplicity. However, **Modern C++ (C++20/23)** provides powerful abstractions that make it an excellent language for implementing interpreters and language runtimes.
In this booklet, however, C++ is not used as a class-heavy object-oriented language. It is used as a **systems language with stronger tools than C**, especially in areas such as type safety, memory management, container support, and expressive value modeling.

### 1.7.1 Modern C++ Advantages in a C-Style Design

Table 7-3: Modern C++ Features in C-Style Interpreter Design

| Feature | Usage in Interpreter Implementation |
|---|---|
| `std::variant` | Represent different AST node types safely without class hierarchies |
| `std::optional` | Represent parsing failures or missing values |
| `std::string_view` | Efficient source text processing in the lexer |
| `std::unique_ptr` | Safe ownership of recursive AST nodes |
| `std::vector` | Natural storage for tokens, statements, and blocks |
| `std::unordered_map` | Efficient symbol table for variables and environments |

**Example: AST Node Representation Without OOP**

```
struct Number {
    double value;
};

struct Variable {
    std::string name;
};

struct BinaryOp {
    std::string op;
```

```
    std::unique_ptr<Expr> left;
    std::unique_ptr<Expr> right;
};

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

This design stays **type-safe**, **explicit**, and **close to classic systems programming style**, while still taking advantage of the strongest features of Modern C++.

## 1.8 REPL: Read–Eval–Print Loop

A REPL is an interactive environment where users can type code and immediately see the result.
The acronym stands for:

- Read
- Evaluate
- Print
- Loop

Many popular languages rely heavily on REPL environments for experimentation and debugging.

### 1.8.1 REPL Execution Flow

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

### 1.8.2 Example REPL Session

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

## 1.9 What You Will Learn

By the end of this booklet, you will understand:

- How raw source code is transformed into tokens
- How grammars are parsed into structured syntax trees
- How AST nodes are evaluated inside a runtime environment
- How interpreters manage variables and program state
- How to structure and extend a language implementation
- How Modern C++ can be used in a procedural, C-style architecture

## 1.10 Who This Guide Is For

This guide is intended for:

- **C and C++ programmers** who want to explore language implementation
- **Students** studying compilers or programming language design
- **Developers** interested in building DSLs or scripting systems

- **Educators** teaching language internals

It is especially suitable for readers who prefer **explicit procedural design** over class-based abstractions.

## 1.11 Summary

This chapter introduced the fundamental ideas behind interpreters, explained how they differ from compilers, and outlined the key phases involved in interpreter execution.
It also established the central philosophy of this booklet: building a small interpreter in **Modern C++ with a C-style architecture**, avoiding object-oriented design while still benefiting from smart pointers, STL containers, and strong type-safe language features.
In the following chapters, we will progressively construct a working interpreter from the ground up using this procedural Modern C++ approach.

# Chapter 2

# Project Setup

## 2.1 Introduction

Before implementing any interpreter components, it is important to prepare a clean and scalable project environment. A well-organized structure and reliable build system will make the development process significantly easier as the interpreter grows in complexity.
In this chapter we establish the technical foundation of the interpreter project. Specifically, we will cover:

- Organizing the project directory and source files
- Setting up a Modern C++ toolchain using CMake
- Writing and compiling a minimal test program
- Creating the entry point for the interpreter's REPL environment

The goal is to build a project structure that can evolve naturally as we later add the lexer, parser, abstract syntax tree (AST), runtime environment, and evaluator.

Although the implementation is written in Modern C++, the project follows a **procedural C-style architecture**. The interpreter components are implemented primarily using `struct` types and free functions instead of object-oriented class hierarchies.

## 2.2 Directory and File Structure

A clear and modular file layout is essential for interpreter development. Each subsystem of the interpreter should reside in a dedicated file so that responsibilities remain well separated.
The following structure will be used throughout this guide:

```
interpreter/
  CMakeLists.txt
  main.cpp
  repl.hpp
  repl.cpp
  lexer.hpp
  lexer.cpp
  parser.hpp
  parser.cpp
  ast.hpp
  interpreter.hpp
  interpreter.cpp
  tokens.hpp
  utils.hpp
```

This structure reflects the logical architecture of the interpreter.

Table 2-1: Source File Responsibilities in Interpreter Project

| File | Responsibility |
|---|---|
| `main.cpp` | Program entry point and interpreter startup logic |
| `repl.*` | Implementation of the Read–Eval–Print Loop (REPL) |
| `lexer.*` | Lexical analysis: converts raw source code into tokens |
| `parser.*` | Converts tokens into an Abstract Syntax Tree (AST) |
| `ast.hpp` | Definitions of AST node structures |
| `interpreter.*` | Executes and evaluates the AST nodes |
| `tokens.hpp` | Token type definitions and token structure |
| `utils.hpp` | Helper utilities (formatting, diagnostics, helpers) |

This modular approach allows each interpreter subsystem to evolve independently.

## 2.3 Installing Required Tools

Before building the interpreter, ensure that your development environment includes a modern C++ compiler and CMake.

### 2.3.1 Modern C++ Compiler

To compile code written in C++20 or C++23, use a reasonably recent compiler version.

| Compiler | Recommended Minimum Version |
|---|---|
| GCC | 10.3 or newer (11+ recommended) |
| Clang | 12 or newer |
| MSVC | Visual Studio 2019 (16.9) or later |

### 2.3.2 CMake

CMake is used as the cross-platform build system for the interpreter.

A version of **3.20 or later** is recommended.

Typical installation methods include:

**Linux**

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

**macOS**

```
brew install cmake
```

**Windows**

CMake can be installed through the Visual Studio Installer or downloaded from the official CMake distribution.

## 2.4 Writing `CMakeLists.txt`

Create the following file in the project root.

### Create File: `CMakeLists.txt`

```
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` generates a `compile_commands.json` file useful for tools such as `clangd`.
- As the interpreter grows, this configuration can be extended to include libraries, tests, or multiple executable targets.

## 2.5 Hello World Test

Before implementing interpreter logic, verify that the toolchain and build system are functioning correctly.

### Create File: `main.cpp`

Create the following minimal program:

```
#include <iostream>

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

Build the project:

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

Run the executable:

```
./interpreter
```

Expected output:

```
Simple Interpreter Ready
```

If this message appears, the build system and compiler are correctly configured.

## 2.6 Creating the REPL Entry Point

The interpreter will operate interactively using a REPL (Read–Eval–Print Loop). The REPL reads user input, evaluates it, and prints the result.

## Update File: `main.cpp`

Replace the previous test program with the following code:

```
#include "repl.hpp"

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

## Create File: `repl.hpp`

```
#pragma once

void start_repl();
```

## Create File: `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 the interpreter.

Example session:

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

Later chapters will connect this REPL to the lexer, parser, and evaluator.

## 2.7 Optional IDE Integration

Modern IDEs provide strong integration with CMake projects.

### 2.7.1 Visual Studio Code

Recommended extensions:

- CMake Tools
- C/C++ IntelliSense
- clangd (recommended language server)

Example configuration in `.vscode/settings.json`:

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

### 2.7.2 CLion

CLion automatically detects `CMakeLists.txt`. The C++ standard can be configured directly in the CMake settings panel.

## 2.8 Summary

At this point, the interpreter project has a stable development foundation:

- A modular directory layout
- A portable CMake build system
- A working compiler toolchain
- A minimal REPL entry point

This structure will support all future interpreter components including the lexer, parser, AST, evaluation engine, and control-flow execution.

## 2.9 Exercises

1. **Modularize Headers**

   Ensure that all function declarations are placed in `.hpp` files and protected using `#pragma once`.

2. **Add a Logging Utility**

   Example declaration for a utility function:

```cpp
// File: utils.hpp
void log(const std::string& message);
```

3. **Add Version Information**

   Example constant declaration:

   ```
   // Example location: main.cpp
   constexpr const char* VERSION = "0.1.0";
   ```

4. **Build Modes**

   Experiment with different build configurations:

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

   Compare the behavior of Debug and Release builds.

# Chapter 3

# Lexer (Tokenizer)

## 3.1 What Is Lexical Analysis?

Lexical analysis is the first stage of interpreter execution. In this phase, the raw source code—typically represented as a sequence of characters—is scanned and converted into a structured sequence of **tokens**.

Tokens represent the smallest meaningful elements of the language, such as numbers, identifiers, operators, punctuation symbols, and keywords. The lexer simplifies the job of the parser by transforming the character stream into a structured token stream.

**Example Input**

```
x = 3 + 4
```

**Expected Tokens**

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

Once the source code has been transformed into tokens, the parser can analyze the syntactic structure of the program.

## 3.2 Token Type Definition

We begin by defining the set of token types that the lexer can recognize.

### Update File: `tokens.hpp`

Replace the previous contents of `tokens.hpp` with the following definitions:

```
#pragma once

#include <string_view>

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 Design Notes

- `std::string_view` allows the lexer to reference substrings of the original source code without allocating new strings.
- Tracking the line and column positions greatly improves diagnostic messages and debugging capabilities.

## 3.3 Designing the Lexer Structure

In keeping with the procedural C-style architecture of this interpreter, the lexer is implemented using a `struct` and a set of free functions rather than a class with member methods.
The lexer structure stores the state of the scanning process.

### Update File: `lexer.hpp`

Replace the previous contents of `lexer.hpp` with the following definitions:

```
#pragma once

#include <string_view>
#include <vector>
#include "tokens.hpp"
```

```
struct Lexer {

    std::string_view source;

    size_t start = 0;
    size_t current = 0;

    int line = 1;
    int column = 1;
};

std::vector<Token> tokenize(Lexer* lexer);
```

This structure keeps the lexer state explicit and easy to reason about.

## 3.4 Lexer Helper Functions

Helper functions operate directly on the lexer structure.

### Update File: `lexer.cpp`

```
#include "lexer.hpp"
#include <cctype>

static bool is_at_end(const Lexer* lexer) {
    return lexer->current >= lexer->source.size();
}

static char peek(const Lexer* lexer) {

    if (is_at_end(lexer))
```

```
        return '\0';

    return lexer->source[lexer->current];
}

static char advance(Lexer* lexer) {

    char c = peek(lexer);

    ++lexer->current;
    ++lexer->column;

    return c;
}
```

These functions allow the lexer to safely inspect and consume characters from the input stream.

## 3.5 Skipping Whitespace

Whitespace characters are not meaningful tokens in most languages, so the lexer skips them.

### Update File: `lexer.cpp`

```
static void skip_whitespace(Lexer* lexer) {

    while (!is_at_end(lexer)) {

        char c = peek(lexer);

        if (c == ' ' || c == '\t' || c == '\r') {
```

```
            advance(lexer);
        }
        else if (c == '\n') {

            advance(lexer);

            lexer->line++;
            lexer->column = 1;
        }
        else {
            break;
        }
    }
}
```

Tracking newline characters ensures that token positions remain accurate.

## 3.6 Reading Numeric Literals

Numeric tokens are sequences of digits.

### Update File: `lexer.cpp`

```
static Token read_number(Lexer* lexer) {

    size_t number_start = lexer->current;

    while (std::isdigit(peek(lexer)))
        advance(lexer);

    auto lexeme =
```

```
        lexer->source.substr(number_start,
                             lexer->current - number_start);

    return Token{
        TokenType::Number,
        lexeme,
        lexer->line,
        lexer->column
    };
}
```

In a more advanced interpreter, this function could also support floating-point numbers or scientific notation.

## 3.7 Reading Identifiers and Keywords

Identifiers represent variable names or user-defined symbols. Keywords represent reserved words in the language.

### Update File: `lexer.cpp`

```
static Token read_identifier_or_keyword(Lexer* lexer) {

    size_t id_start = lexer->current;

    while (std::isalnum(peek(lexer)) || peek(lexer) == '_')
        advance(lexer);

    auto text =
        lexer->source.substr(id_start,
                             lexer->current - id_start);
```

```
    if (text == "print")
        return Token{TokenType::KeywordPrint, text, lexer->line, lexer->column};

    if (text == "if")
        return Token{TokenType::KeywordIf, text, lexer->line, lexer->column};

    if (text == "while")
        return Token{TokenType::KeywordWhile, text, lexer->line, lexer->column};

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

This mechanism allows the lexer to distinguish reserved keywords from ordinary identifiers.

## 3.8 Reading Operators and Symbols

Operators and punctuation symbols are recognized individually.

### Update File: `lexer.cpp`

```
static Token read_operator_or_symbol(Lexer* lexer) {

    char c = advance(lexer);

    switch (c) {

        case '+':
        case '-':
        case '*':
        case '/':
            return Token{
```

```
            TokenType::Operator,
            lexer->source.substr(lexer->current - 1, 1),
            lexer->line,
            lexer->column
        };

    case '=':
        return Token{
            TokenType::Assign,
            lexer->source.substr(lexer->current - 1, 1),
            lexer->line,
            lexer->column
        };

    case '(':
        return Token{TokenType::LeftParen, "(", lexer->line, lexer->column};

    case ')':
        return Token{TokenType::RightParen, ")", lexer->line, lexer->column};

    case '{':
        return Token{TokenType::LeftBrace, "{", lexer->line, lexer->column};

    case '}':
        return Token{TokenType::RightBrace, "}", lexer->line, lexer->column};

    case ';':
        return Token{TokenType::Semicolon, ";", lexer->line, lexer->column};

    default:
        return Token{
            TokenType::Unknown,
            lexer->source.substr(lexer->current - 1, 1),
```

```
                lexer->line,
                lexer->column
            };
    }
}
```

## 3.9 The Tokenization Loop

The main tokenization loop repeatedly scans characters and produces tokens until the end of the input is reached.

### Update File: `lexer.cpp`

```
std::vector<Token> tokenize(Lexer* lexer) {

    std::vector<Token> tokens;

    while (!is_at_end(lexer)) {

        skip_whitespace(lexer);

        if (is_at_end(lexer))
            break;

        char c = peek(lexer);

        if (std::isdigit(c)) {

            tokens.push_back(read_number(lexer));
        }
        else if (std::isalpha(c) || c == '_') {
```

```
            tokens.push_back(read_identifier_or_keyword(lexer));
        }
        else {

            tokens.push_back(read_operator_or_symbol(lexer));
        }
    }

    tokens.push_back(Token{
        TokenType::EndOfFile,
        "",
        lexer->line,
        lexer->column
    });

    return tokens;
}
```

The `EndOfFile` token signals the end of the input stream and simplifies parser logic.

## 3.10 Example Output

Given the input:

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

The generated token stream would resemble:

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

## 3.11 Debug Utility: Printing Tokens

During development it is often helpful to inspect the generated token stream.

### Create or Update File: `utils.hpp`

You may place the following helper function in `utils.hpp`:

```
#pragma once

#include <iostream>
#include <vector>
#include "tokens.hpp"

inline 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";
    }
}
```

This function can be integrated into the REPL to visualize lexer output during debugging.

## 3.12 Exercises

1. **Add Support for Comments**

   Extend the lexer to ignore comments beginning with:

   ```
   // comment text
   ```

   The lexer should skip characters until the end of the line.

2. **Add String Literals**

   Support string tokens enclosed in quotes:

   ```
   "hello world"
   ```

   Add a new token type `String` in `tokens.hpp` and extend the lexer accordingly.

3. **Add Multi-Character Operators**

   Extend operator recognition to support:

   ```
   ==  !=  <=  >=
   ```

   This requires a lookahead mechanism in the lexer.

4. **Track Source Ranges**

   Enhance the `Token` structure to include:

   ```
   size_t start_pos;
   size_t end_pos;
   ```

   These fields allow the interpreter to highlight exact source ranges when reporting errors.

# Chapter 4

# Writing the Parser

## 4.1 What Is a Parser?

A parser is the second major phase of an interpreter. It receives the **token stream** produced by the lexer and transforms it into a structured representation of the program.
This structure is called the **Abstract Syntax Tree (AST)**.
Each node of the AST represents a meaningful construct in the language, such as:

- Numeric literals
- Variables
- Binary operations
- Assignments
- Statements such as `print`, `if`, or `while`

The AST captures the logical structure of the program while removing unnecessary syntactic details.

### 4.1.1 Input vs Output

| Component | Input | Output |
|---|---|---|
| Lexer | Raw source code | Token sequence |
| Parser | Token sequence | AST (Abstract Syntax Tree) |
| Evaluator | AST | Runtime result or effect |

## 4.2 Why Recursive Descent?

In this interpreter we use **recursive descent parsing**. This approach is widely used in small language implementations because:

- It is straightforward to implement
- It maps directly to the grammar rules
- The resulting code is easy to read and maintain

For languages with simple grammars, recursive descent provides a clean and flexible parsing strategy.

## 4.3 Basic Grammar of the Language

```
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 ")"
```

This grammar also defines operator precedence.

## 4.4 AST Node Recap

### Update File: `ast.hpp`

```
#pragma once

#include <memory>
#include <string>
#include <vector>
```

```
#include <variant>

struct Number {
    double value;
};

struct Variable {
    std::string name;
};

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

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

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

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

```
struct If {
    std::unique_ptr<struct Expr> condition;
    Block body;
};

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

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

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

## 4.5 Parser State Structure

Instead of implementing the parser as a class, we use a simple structure holding the parser state.

## Update File: parser.hpp

```
#pragma once

#include <vector>
#include "tokens.hpp"
#include "ast.hpp"

struct Parser {

    const std::vector<Token>* tokens;

    size_t current = 0;
};

std::vector<Stmt> parse(Parser* parser);
\end{minted}

\section{Parser Helper Functions}

\subsection*{Update File: \texttt{parser.cpp}}

\begin{minted}{cpp}
#include "parser.hpp"
#include <stdexcept>

static Token peek(Parser* p) {
    return (*p->tokens)[p->current];
}
```

```
static bool is_at_end(Parser* p) {
    return peek(p).type == TokenType::EndOfFile;
}

static Token advance(Parser* p) {

    if (!is_at_end(p))
        p->current++;

    return (*p->tokens)[p->current - 1];
}

static bool check(Parser* p, TokenType type) {

    if (is_at_end(p))
        return false;

    return peek(p).type == type;
}

static bool match(Parser* p, TokenType type) {

    if (check(p, type)) {

        advance(p);
        return true;
    }
```

```
    return false;
}
```

## 4.6 Parsing Expressions

### Update File: `parser.cpp`

```
static Expr parse_factor(Parser* p);

static Expr parse_term(Parser* p) {

    Expr expr = parse_factor(p);

    while (check(p, TokenType::Operator) &&
          (peek(p).lexeme == "*" || peek(p).lexeme == "/")) {

        std::string op = std::string(advance(p).lexeme);

        Expr right = parse_factor(p);

        expr = BinaryOp{
            op,
            std::make_unique<Expr>(std::move(expr)),
            std::make_unique<Expr>(std::move(right))
        };
    }

    return expr;
}
```

```
static Expr parse_expression(Parser* p) {
```

```
    Expr expr = parse_term(p);

    while (check(p, TokenType::Operator) &&
           (peek(p).lexeme == "+" || peek(p).lexeme == "-")) {

        std::string op = std::string(advance(p).lexeme);

        Expr right = parse_term(p);

        expr = BinaryOp{
            op,
            std::make_unique<Expr>(std::move(expr)),
            std::make_unique<Expr>(std::move(right))
        };
    }

    return expr;
}
```

```
static Expr parse_factor(Parser* p) {

    if (match(p, TokenType::Number)) {

        double value =
            std::stod(std::string((*p->tokens)[p->current - 1].lexeme
                ));

        return Number{value};
    }

    if (match(p, TokenType::Identifier)) {
```

```
        return Variable{
            std::string((*p->tokens)[p->current - 1].lexeme)
        };
    }

    if (match(p, TokenType::LeftParen)) {

        Expr expr = parse_expression(p);

        if (!match(p, TokenType::RightParen))
            throw std::runtime_error("Expected ')'");

        return expr;
    }

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

## 4.7 Parsing Statements

### Update File: `parser.cpp`

```
static Block parse_block(Parser* p);

static Stmt parse_statement(Parser* p) {

    if (match(p, TokenType::KeywordPrint)) {
```

```
        Expr value = parse_expression(p);

        return Print{
            std::make_unique<Expr>(std::move(value))
        };
    }

    if (check(p, TokenType::Identifier) &&
        (*p->tokens)[p->current + 1].type == TokenType::Assign) {

        std::string name =
            std::string(advance(p).lexeme);

        advance(p);

        Expr value = parse_expression(p);

        return Assignment{
            name,
            std::make_unique<Expr>(std::move(value))
        };
    }

    if (match(p, TokenType::KeywordIf)) {

        Expr cond = parse_expression(p);

        Block body = parse_block(p);
```

```
        return If{
            std::make_unique<Expr>(std::move(cond)),
            body
        };
    }

    if (match(p, TokenType::KeywordWhile)) {

        Expr cond = parse_expression(p);

        Block body = parse_block(p);

        return While{
            std::make_unique<Expr>(std::move(cond)),
            body
        };
    }

    Expr expr = parse_expression(p);

    return Assignment{
        "_",
        std::make_unique<Expr>(std::move(expr))
    };
}
```

## 4.8 Parsing Blocks

### Update File: parser.cpp

```
static Block parse_block(Parser* p) {

    if (!match(p, TokenType::LeftBrace))
        throw std::runtime_error("Expected '{'");

    std::vector<Stmt> statements;

    while (!check(p, TokenType::RightBrace) &&
           !is_at_end(p)) {

        statements.push_back(parse_statement(p));
    }

    if (!match(p, TokenType::RightBrace))
        throw std::runtime_error("Expected '}'");

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

## 4.9 Parsing the Program

### Update File: parser.cpp

```
std::vector<Stmt> parse(Parser* p) {

    std::vector<Stmt> statements;
```

```
    while (!is_at_end(p)) {

        statements.push_back(parse_statement(p));
    }

    return statements;
}
```

## 4.10 Example: Source Code to AST

Input program:

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

Simplified AST representation:

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

Print(
    Variable("x")
)
```

## 4.11 Summary

In this chapter you implemented a procedural recursive descent parser that:

- Converts token sequences into an Abstract Syntax Tree
- Implements expression precedence rules
- Supports assignments, print statements, conditionals, and loops
- Uses a simple C-style architecture with structs and functions

The AST generated by the parser will be executed by the interpreter runtime in the next chapter.

# Chapter 5

# AST Node Definitions and Tree Design

After parsing the token stream, the interpreter must represent the program in a structured form that can be evaluated efficiently. This representation is known as the **Abstract Syntax Tree (AST)**.
An AST is a hierarchical tree structure where each node corresponds to a syntactic construct of the language. Unlike the raw token sequence, the AST captures the logical structure of the program rather than its textual form.
For example, arithmetic expressions, assignments, control-flow statements, and function calls can all be represented as nodes within the tree.

## 5.1 Goals of This Chapter

The objectives of this chapter are:

- Define the core AST node structures for expressions and statements.

- Represent node types using `std::variant`.
- Implement safe ownership semantics using `std::unique_ptr`.
- Prepare the AST for efficient traversal during evaluation.

This interpreter follows a **procedural C-style architecture**. The AST is built using simple `struct` types and standard containers rather than class hierarchies.

## 5.2 Expression Node Types

Expressions represent computations that produce values. In our small language we support the following expression categories:

- **Number**: literal numeric values such as `42`
- **Variable**: identifier references such as `x`
- **BinaryOp**: arithmetic operations such as `a + b`
- **Assignment**: variable assignment such as `x = 10`

### Update File: `ast.hpp`

```
#pragma once

#include <memory>
#include <string>
#include <vector>
#include <variant>

struct Number {
```

```
    double value;
};

struct Variable {
    std::string name;
};

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

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

The children of expression nodes are stored using `std::unique_ptr`. This clearly expresses ownership and ensures that recursive tree structures are automatically destroyed when nodes go out of scope.

## 5.3 Statement Node Types

Statements represent actions rather than values. In this interpreter we support the following statement types:

- **Print**: output a value
- **If**: conditional execution
- **While**: loop execution

- **Block**: a group of statements

### Update File: `ast.hpp`

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

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

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

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

Blocks allow nested program structures such as loops and conditional branches.

## 5.4 Combining Node Types with `std::variant`

To store different AST node types safely in a single container we use `std::variant`. This provides a type-safe discriminated union.

### 5.4.1 Expression Variant

### Update File: `ast.hpp`

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

### 5.4.2 Statement Variant

### Update File: `ast.hpp`

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

This allows the parser and interpreter to store heterogeneous node types in a single container.

## 5.5 Why Use `std::variant`?

Traditional compilers often use class hierarchies and virtual methods for AST nodes. Modern C++ provides a simpler alternative using `std::variant`.
Advantages include:

- Strong compile-time type safety

- No dynamic casting
- Clear and explicit node types
- Efficient memory layout

Node dispatch is handled using `std::visit`.

### 5.5.1 Example Dispatch Using `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_variable(a);
    }

}, expr);
```

This pattern allows the interpreter to process different AST node types without relying on virtual functions.

## 5.6 Memory Management with `std::unique_ptr`

Recursive structures such as AST nodes require careful memory management.
Each AST node owns its child nodes through `std::unique_ptr`.
Benefits include:

- Automatic memory cleanup
- Clear ownership semantics
- Safe recursive structures

### 5.6.1 Example

```
struct BinaryOp {

    std::string op;

    std::unique_ptr<Expr> left;

    std::unique_ptr<Expr> right;
};
```

When the parent node is destroyed, the entire subtree is automatically freed.

## 5.7 Example AST Structure

Consider the source code:

```
x = 1 + 2
```

The AST representation would resemble:

```
Assignment{
    "x",
    std::make_unique<Expr>(
        BinaryOp{
            "+",
            std::make_unique<Expr>(Number{1}),
            std::make_unique<Expr>(Number{2})
        }
    )
}
```

Conceptually, the tree structure looks like:

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

During evaluation, the interpreter recursively traverses this tree to compute the result.

## 5.8 Extending the AST

As the language evolves, the AST can be extended with additional node types such as:

- Function declarations and calls
- Boolean expressions
- Logical operators
- Return statements
- Break and continue statements

Each new construct can be added as a new `struct` and included in the appropriate `std::variant`.

## 5.9 Possible Enhancements

Several improvements can make the AST system more powerful:

- Store source location information in AST nodes for diagnostics
- Implement debugging utilities to visualize AST structures
- Introduce visitor utilities to simplify tree traversal
- Use `std::expected` (C++23) for safer AST construction

## 5.10 Summary

In this chapter you:

- Defined the core AST node structures used by the interpreter
- Represented heterogeneous node types using `std::variant`
- Used `std::unique_ptr` for safe recursive ownership
- Built a tree representation suitable for evaluation

In the next chapter we will implement the **interpreter engine** that traverses the AST and executes the program.

# Chapter 6

# Evaluator – Walking the AST to Run Code

After parsing the source code into an Abstract Syntax Tree (AST), the final stage of interpretation is **evaluation**. In this phase the interpreter walks through the AST and performs the operations described by each node.
Evaluation transforms the static structure of the program into dynamic behavior: arithmetic is computed, variables are updated, and control flow is executed.

## 6.1 What Is Evaluation?

The evaluator (or interpreter runtime) is responsible for:

- Traversing expression and statement nodes.
- Computing numeric values.
- Managing variables and program state.

- Performing side effects such as `print()`.
- Executing control-flow constructs such as `if` and `while`.

The evaluator essentially acts as a small virtual machine that executes the AST representation of the program.

## 6.2 Evaluation Strategy

Our interpreter evaluates nodes using the following techniques:

- `std::variant` combined with `std::visit` for type-safe dispatch.
- Recursive traversal of child nodes.
- A **symbol table** implemented with `std::unordered_map` to store variables.

This procedural design keeps the interpreter simple while still taking advantage of Modern C++ safety features.

## 6.3 Symbol Table – Storing Variables

To store variable values during execution we use a global symbol table.

### Update File: `interpreter.cpp`

```
#include <unordered_map>
#include <string>

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

Each variable name maps to its current numeric value.
In more advanced interpreters this table is replaced by a stack of **environments** to support nested scopes and functions.

## 6.4 Evaluating Expressions

Expression nodes produce values. The interpreter evaluates them recursively.

### Update File: `interpreter.hpp`

```
#pragma once

#include "ast.hpp"

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

### Update File: `interpreter.cpp`

```
#include "interpreter.hpp"
#include <stdexcept>

double evaluate_expr(const Expr& expr)
{
    return std::visit(overloaded{

        [](const Number& n)
        {
            return n.value;
        },

        [](const Variable& v)
```

```
        {
            if (!globals.contains(v.name))
                throw std::runtime_error("Undefined variable: " + v.name);

            return globals[v.name];
        },

        [](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 value = evaluate_expr(*a.value);
            globals[a.name] = value;
            return value;
        }

    }, expr);
}
```

This function uses `std::visit` to dispatch behavior based on the actual expression type stored inside the variant.

## 6.5 Executing Statements

Statements perform actions rather than producing values.

### Update File: `interpreter.hpp`

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

### Update File: `interpreter.cpp`

```
#include <iostream>

void execute_stmt(const Stmt& stmt)
{
    std::visit(overloaded{

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

        [](const Assignment& a)
        {
            evaluate_expr(a);
        },

        [](const Block& block)
        {
```

```
            for (const auto& s : block.statements)
            {
                execute_stmt(s);
            }
        },

        [](const If& i)
        {
            if (evaluate_expr(*i.condition) != 0.0)
            {
                for (const auto& s : i.body)
                    execute_stmt(s);
            }
        },

        [](const While& w)
        {
            while (evaluate_expr(*w.condition) != 0.0)
            {
                for (const auto& s : w.body)
                    execute_stmt(s);
            }
        }

    }, stmt);
}
```

In this interpreter, any non-zero numeric value is considered **true**, while `0.0` represents **false**.

## 6.6 Example: Evaluating a Program

Consider the following program:

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

The interpreter performs the following steps:

1. Evaluate the expression `3 + 4`, producing the value `7`.
2. Store `x = 7` in the symbol table.
3. Execute the `print(x)` statement and output the value.

## 6.7 Improving the Runtime Design

As the language grows, the evaluator can be improved with additional abstractions. Possible improvements include:

- Implementing an **environment stack** to support nested scopes.
- Separating expression evaluation and statement execution into dedicated modules.
- Adding debugging or tracing facilities.
- Providing structured runtime error messages.

## 6.8 Extending the Value System

Currently the interpreter supports only numeric values. A more complete language may introduce a general value representation.

### Example Future Extension

```
using Value = std::variant<
    double,
    bool,
    std::string
>;
```

This would allow support for:

- Boolean expressions
- String literals
- More advanced language constructs

Comparison operators could then return boolean values:

```
==  !=  <  >  <=  >=
```

Logical operators may also be introduced:

```
&&  ||  !
```

## 6.9 Summary

In this chapter you implemented the core interpreter runtime.

- Expressions are evaluated recursively by traversing the AST.
- Variables are stored and retrieved using a symbol table.
- Statements perform actions such as printing and control flow.

- Execution uses Modern C++ features including `std::variant`, `std::visit`, and smart pointers.

With the evaluator implemented, the interpreter is now capable of executing real programs written in the language.

# Chapter 7

# Statements and REPL – Making Your Language Interactive

At this point the interpreter can tokenize source code, parse it into an AST, and evaluate the resulting structure. The next step is to make the language interactive by implementing a **REPL** – a Read–Eval–Print Loop.
A REPL allows users to enter code interactively and immediately see the results. Many modern languages such as Python, Lua, and JavaScript provide REPL environments for experimentation and rapid development.

## 7.1 What Is a REPL?

REPL stands for:

- **Read** – Accept a line of source code from the user.
- **Eval** – Tokenize, parse, and evaluate the input.

- **Print** – Display the result or any errors.
- **Loop** – Repeat the process until the user exits.

This interactive cycle allows programs to be executed incrementally rather than as a complete file.
REPL environments are particularly useful for:

- Debugging language features
- Rapid experimentation
- Interactive learning
- Building scripting environments

## 7.2 Supporting the `print(expr)` Statement

To make the interpreter useful in an interactive setting, we add a `print` statement that outputs expression values.

### 7.2.1 AST Node Definition

### Update File: `ast.hpp`

Ensure the following node exists in `ast.hpp`.

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

Include this node in the statement variant.

### Update File: `ast.hpp`

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

## 7.3 Parser Support

Extend the parser to recognize the `print` keyword.

### Update File: `parser.cpp`

Add the following logic inside the `parse_statement()` function.

```
Stmt Parser::parse_statement()
{
    if (match(TokenType::KeywordPrint))
    {
        Expr value = parse_expression();

        return Print{
            std::make_unique<Expr>(std::move(value))
        };
    }

    // other statement types handled here
}
```

## 7.4 Evaluator Support

Add support for the `print` statement inside the interpreter.

### Update File: `interpreter.cpp`

Add the following visitor case inside the `execute_stmt` visitor.

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

The interpreter can now display computed values during execution.

## 7.5 Building the REPL Loop

The REPL repeatedly reads input, interprets it, and prints the result.

### Update File: `repl.cpp`

Replace or extend the implementation of `start_repl()`.

```
#include <iostream>
#include <string>

#include "lexer.hpp"
#include "parser.hpp"
#include "interpreter.hpp"

void start_repl()
```

```
{
    std::string line;

    while (true)
    {
        std::cout << ">> ";

        if (!std::getline(std::cin, line))
            break;

        if (line.empty())
            continue;

        try
        {
            Lexer lexer(line);
            auto tokens = lexer.tokenize();

            Parser parser(tokens);
            auto statements = parser.parse();

            for (const auto& stmt : statements)
                execute_stmt(stmt);
        }
        catch (const std::exception& e)
        {
            std::cerr << "Error: "
                      << e.what()
                      << std::endl;
        }
    }
}
```

## 7.6 Program Entry Point

Add the REPL entry point to the program.

### Update File: `main.cpp`

```
#include <iostream>
#include "repl.hpp"

int main()
{
    std::cout << "Simple Interpreter v0.1\n";

    start_repl();

    return 0;
}
```

## 7.7 Example REPL Session

Example interaction with the interpreter:

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

Each line is interpreted independently while the variable environment persists across evaluations.

## 7.8 Improving the REPL

A production-quality REPL can support many additional features.
Possible improvements include:

- **Multi-line input**

  Detect unmatched parentheses or braces to allow multi-line code blocks.

- **Built-in commands**

  Add interpreter commands such as:

  ```
  :exit
  :vars
  :help
  ```

- **Command history**

  Store previous commands using libraries such as `linenoise` or `readline`.

- **Auto-completion**

  Suggest variable names or keywords interactively.

## 7.9 Why REPLs Matter

Interactive environments are a defining feature of many modern languages.
Benefits include:

- Immediate feedback during development
- Rapid experimentation

- Easier debugging of language features
- Convenient scripting environments

REPL systems also provide a valuable development tool while designing the language itself.

## 7.10 Organizing the REPL in the Codebase

To keep the interpreter modular, place REPL logic in its own files.

### Create File: `repl.hpp`

```
#pragma once

void start_repl();
```

### Create or Update File: `repl.cpp`

```
#include "repl.hpp"

void start_repl()
{
    // REPL implementation
}
```

This keeps the interactive environment separate from the interpreter core.

## 7.11 Summary

In this chapter you added interactive capabilities to the interpreter.

- Implemented the `print()` statement.
- Built a REPL loop that reads, evaluates, and prints results.
- Enabled users to run code interactively.
- Structured the interpreter to support future enhancements.

With the REPL in place, your language can now function as a fully interactive scripting environment.

# Chapter 8

# Control Flow – If and While Statements

So far, the language supports variables, arithmetic expressions, and output through `print`. To make the language more expressive, we must introduce **control flow constructs**.
Control flow allows programs to:

- Execute code conditionally (`if`)
- Repeat actions (`while`)

These constructs are fundamental to any programming language because they allow programs to react to changing conditions and perform repetitive computations.

## 8.1 The `if` Statement

### 8.1.1 Syntax

The basic form of the conditional statement is:

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

Optionally, an `else` branch may be provided:

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

The condition expression is evaluated at runtime. If the result is non-zero, the `then` branch is executed.

## 8.2 AST Node for `if`

### Update File: `ast.hpp`

```
struct If {

    std::unique_ptr<Expr> condition;

    std::vector<Stmt> then_branch;

    std::vector<Stmt> else_branch;
};
```

This node stores the conditional expression and both possible execution branches.

## 8.3 Parser Support for `if`

The parser must recognize conditional statements and build the corresponding AST node.

### Update File: `parser.cpp`

```
Stmt Parser::parse_if()
{
    Expr condition = parse_expression();

    auto then_branch = parse_block();

    std::vector<Stmt> else_branch;

    if (match(TokenType::KeywordElse))
    {
        else_branch = parse_block();
    }

    return If{
        std::make_unique<Expr>(std::move(condition)),
        std::move(then_branch),
        std::move(else_branch)
    };
}
```

The parser reads the condition expression and then parses the block representing the body of the conditional statement.

## 8.4 The `while` Loop

Loops allow repeated execution while a condition remains true.

### 8.4.1 Syntax

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

The condition is evaluated before each iteration. If the condition evaluates to zero, the loop terminates.

## 8.5 AST Node for `while`

### Update File: `ast.hpp`

```
struct While {

    std::unique_ptr<Expr> condition;

    std::vector<Stmt> body;
};
```

## 8.6 Parser Support for `while`

### Update File: `parser.cpp`

```
Stmt Parser::parse_while()
{
```

```
    Expr condition = parse_expression();

    auto body = parse_block();

    return While{
        std::make_unique<Expr>(std::move(condition)),
        std::move(body)
    };
}
```

The parser consumes the loop condition and then parses the block of statements representing the loop body.

## 8.7 Parsing Code Blocks

Control flow constructs typically contain blocks of statements enclosed in braces.

### 8.7.1 Block Syntax

```
{
    statement1
    statement2
}
```

### 8.7.2 Block Parsing

### Update File: `parser.cpp`

```
std::vector<Stmt> Parser::parse_block()
{
    if (!match(TokenType::LeftBrace))
```

```
        throw std::runtime_error("Expected '{'");

    std::vector<Stmt> statements;

    while (!check(TokenType::RightBrace) && !is_at_end())
    {
        statements.push_back(parse_statement());
    }

    if (!match(TokenType::RightBrace))
        throw std::runtime_error("Expected '}'");

    return statements;
}
```

Blocks allow nested control-flow constructs and structured program composition.

## 8.8 Evaluator Support

To execute control-flow constructs, we extend the interpreter's statement execution logic.

### Update File: `interpreter.cpp`

Extend the visitor inside `execute_stmt()`.

```
std::visit(overloaded{

    [&](const If& stmt)
    {
```

```
        if (evaluate_expr(*stmt.condition) != 0.0)
        {
            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) != 0.0)
        {
            for (const auto& s : stmt.body)
                execute_stmt(s);
        }
    }

}, stmt);
```

In this interpreter, numeric values are interpreted as boolean conditions:

- `0.0` represents **false**
- Any non-zero value represents **true**

## 8.9 Example Execution

Consider the following interactive session:

```
>> x = 5
>> if (x > 3) { print(100) }
100

>> y = 0
>> while (y < 3) {
       print(y)
       y = y + 1
   }
0
1
2
```

The interpreter evaluates the conditions and executes the appropriate branches or loop iterations.

## 8.10 Representing Code Blocks

Blocks are represented internally as:

```
std::vector<Stmt>
```

Blocks may appear inside:

- `If` statements
- `While` loops
- Nested blocks

Statements are executed sequentially in the order they appear.

## 8.11 Variable Scoping

As languages grow more complex, variables should be organized using scopes.
One simple strategy is to maintain a stack of environments.

### Example Runtime Structure

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

When entering a block:

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

When leaving a block:

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

Variable lookup proceeds from the most recent scope outward. This allows variable shadowing and structured program organization.

## 8.12 Possible Future Extensions

The control-flow system can be expanded with additional constructs:

- `break` and `continue`
- `for` loops
- `switch` statements
- Short-circuit boolean operators

These features can be implemented by extending both the AST and the interpreter evaluation logic.

## 8.13 Summary

In this chapter you implemented structured control flow in the language.

- Added support for conditional execution using `if`.
- Implemented repetition using `while`.
- Introduced block parsing and nested statement execution.
- Discussed environment-based variable scoping.

With control-flow constructs available, the interpreter now supports branching and looping, making it capable of executing non-trivial programs.

# Chapter 9

# Error Handling

A robust interpreter must handle invalid input gracefully. A single syntax mistake or runtime failure should not terminate the entire program. Instead, the interpreter should report the error clearly and continue running when possible.
In this chapter we introduce structured approaches to error handling in the interpreter using Modern C++ features.

## 9.1 Types of Errors

Errors in interpreters generally fall into two categories:

1. **Syntax Errors**

   These occur during lexical analysis or parsing when the source code violates the language grammar. Examples include:

   - unexpected symbols
   - missing parentheses

- incomplete expressions

2. **Runtime Errors**

   These occur during program execution when evaluation cannot proceed correctly. Examples include:

   - division by zero
   - accessing an undefined variable
   - invalid operator usage

Handling both categories properly greatly improves the usability of the language.

## 9.2 Tracking Source Locations

Meaningful diagnostics require precise source location information.

### 9.2.1 Token Metadata

### Update File: `tokens.hpp`

Ensure the `Token` structure contains source position information.

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

The lexer must update the `line` and `column` fields as it processes characters. This allows error messages to point directly to the location of the problem.

## 9.3 Reporting Syntax Errors

Syntax errors are typically detected by the parser when an unexpected token appears.

### Update File: `parser.cpp`

Add the following helper function.

```
[[noreturn]]
void parser_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
    );
}
```

A common parsing helper function is `consume()`, which verifies that the next token matches the expected type.

### Update File: `parser.cpp`

```
Token consume(std::vector<Token>& tokens,
              size_t& current,
              TokenType expected)
{
    if (tokens[current].type == expected)
        return tokens[current++];
```

```
    parser_error(tokens[current],
        "Unexpected token");
}
```

If the expected token is not present, the parser throws a syntax error.

## 9.4 Runtime Error Handling

Runtime errors occur during evaluation of the AST.
Examples include invalid operations such as division by zero.

### Update File: `interpreter.cpp`

```
double evaluate_binary_op(
    const std::string& op,
    double left,
    double right)
{
    if (op == "/")
    {
        if (right == 0.0)
            throw std::runtime_error(
                "Runtime error: division by zero");

        return left / right;
    }

    if (op == "+") return left + right;
    if (op == "-") return left - right;
    if (op == "*") return left * right;
```

```
    throw std::runtime_error(
        "Unknown operator: " + op);
}
```

Such errors should be caught at a higher level to prevent the program from terminating.

## 9.5 Handling Errors in the REPL

The REPL is an ideal location to catch errors so the interpreter can continue running.

### Update File: `repl.cpp`

```
try {

    Lexer lexer(line);
    auto tokens = lexer.tokenize();

    Parser parser(tokens);
    auto program = parser.parse();

    for (const auto& stmt : program)
        execute_stmt(stmt);

}
catch (const std::exception& ex) {

    std::cerr << "Error: "
              << ex.what()
              << std::endl;
}
```

By catching exceptions at this level, the interpreter remains active and ready for the next command.

## 9.6 Using `std::optional`

Sometimes it is preferable to return a failure value instead of throwing an exception. C++ provides `std::optional` for representing operations that may fail.

### Example Parsing Helper

```
#include <optional>

std::optional<Token>
try_consume(const std::vector<Token>& tokens,
            size_t& current,
            TokenType expected)
{
    if (tokens[current].type == expected)
        return tokens[current++];

    return std::nullopt;
}
```

This approach can simplify parser logic in certain situations.

## 9.7 Custom Result Types

Another approach is to use a structured result type.

### Create File: `utils.hpp`

```
template<typename T>
struct Result {
```

```
    T value;

    std::string error;

    bool ok;
};
```

Example usage:

```
Result<double>
safe_divide(double a, double b)
{
    if (b == 0.0)
        return {0.0, "Division by zero", false};

    return {a / b, "", true};
}
```

This design avoids exceptions and can make error propagation more explicit.

## 9.8 Displaying Errors with Context

Helpful diagnostics often include a visual indicator showing where the error occurred.

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

To support this feature, the interpreter must store the original source line and compute the spacing needed to place the caret under the error position.

## 9.9 Structured Error Types

More advanced interpreters often define custom error structures.

### Example Error Structure

```
struct InterpreterError {

    std::string message;

    int line;
    int column;
};
```

Specialized error types may also be defined:

```
struct SyntaxError  : InterpreterError {};
struct RuntimeError : InterpreterError {};
```

These types can be converted into exceptions when needed.

## 9.10 REPL Integration

The full REPL loop should be protected with error handling so that the interpreter continues running after failures.

### Update File: `repl.cpp`

```
while (true)
{
    try {

        std::string line;

        std::getline(std::cin, line);
```

```
        Lexer lexer(line);
        auto tokens = lexer.tokenize();

        Parser parser(tokens);
        auto program = parser.parse();

        for (const auto& stmt : program)
            execute_stmt(stmt);
    }
    catch (const std::exception& e) {

        std::cerr << "Error: "
                  << e.what()
                  << std::endl;
    }
}
```

This ensures that even severe errors do not terminate the interactive environment.

## 9.11 Summary

In this chapter we implemented the interpreter's error handling system.

- Tokens now track line and column positions.
- The parser reports clear syntax errors.
- Runtime errors are detected during evaluation.
- Exceptions are caught at the REPL level to keep the interpreter running.
- Optional and structured result types provide alternative approaches to error handling.

A clear and reliable error system significantly improves the usability and professionalism of the language.

# Chapter 10

# Final Touches and Next Steps

In this final chapter we summarize the interpreter you have built and discuss how it can evolve into a more capable programming language.
Although the interpreter implemented in this guide is intentionally small, its architecture already reflects the core components used in real language runtimes.
Modern C++ provides powerful tools for building such systems with clarity, type safety, and strong memory guarantees. The design choices used throughout this guide provide a solid foundation for future experimentation and growth.

## 10.1 Recap of the Implemented Features

Over the course of this guide you have constructed a working interpreter from the ground up.
The system now includes the following major components:

- **Lexical Analysis**

Source code is scanned character-by-character and converted into a stream of tokens representing identifiers, numbers, operators, and structural symbols.

- **Recursive Descent Parsing**

  The token stream is processed by a grammar-driven parser that builds an Abstract Syntax Tree (AST).

- **AST Construction**

  Expressions and statements are represented using simple C-style structures combined with `std::variant` and `std::unique_ptr`.

- **Expression Evaluation**

  Arithmetic operations, variable references, and assignments are evaluated by recursively traversing the AST.

- **Control Flow**

  Conditional execution (`if`) and looping (`while`) are supported.

- **Interactive Execution**

  A REPL allows users to enter code line-by-line and immediately observe the results.

- **Error Handling**

  Syntax and runtime errors are detected and reported without terminating the interpreter.

The technologies used throughout this interpreter include:

- `std::variant` for representing heterogeneous AST nodes

- `std::unique_ptr` for safe ownership of recursive structures
- `std::optional` and exceptions for error reporting
- CMake for portable project configuration
- Modern C++20/23 language features

Even in this compact form, the interpreter demonstrates the essential ideas behind real programming language implementations.

## 10.2 Extending the Language

The current interpreter is intentionally minimal, but it can be extended in many ways.

### 10.2.1 User-Defined Functions

Supporting functions introduces several new concepts:

- Function definitions with parameters
- Local variable scope
- Call stack management
- Return values

Example syntax:

```
def add(a, b) {
    return a + b
}

print(add(2, 3))
```

Implementing this feature requires a call stack and a dedicated environment for each function invocation.

### 10.2.2 String Support

The interpreter currently operates only on numeric values. A natural extension is to support strings.

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

## Possible Runtime Representation

```
using Value = std::variant<
    double,
    std::string
>;
```

Supporting strings requires extending:

- token definitions in the lexer
- AST node types
- runtime value representation

### 10.2.3 File Input and Output

File operations make the language useful for automation tasks.

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

### Example Implementation

```
#include <fstream>

std::string read_file(const std::string& path)
{
    std::ifstream file(path);

    if (!file)
        throw std::runtime_error("Cannot open file");

    return std::string(
        std::istreambuf_iterator<char>(file),
        std::istreambuf_iterator<char>()
    );
}
```

These features can be implemented using standard C++ facilities such as `std::filesystem` and file streams.

## 10.3 Scaling the Interpreter

As the language grows, architectural improvements become necessary.

### 10.3.1 Modular Architecture

Larger interpreters benefit from clear separation between subsystems:

- Lexer
- Parser
- AST definitions

- Evaluator
- Runtime environment
- Built-in functions
- REPL interface

This separation improves maintainability and compilation performance.

### 10.3.2 Richer Type System

Future versions of the language may support additional data types:

- Boolean values
- Arrays and lists
- Associative maps
- User-defined structures

## Example Runtime Value Representation

```
using Value = std::variant<
    double,
    bool,
    std::string,
    std::vector<Value>
>;
```

### 10.3.3 Scope Management

Instead of a single global symbol table, interpreters typically maintain a stack of environments representing nested scopes.

### Example Environment Stack

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

This allows:

- Local variables
- Variable shadowing
- Closures

## 10.3.4 Function Call Stack

Function execution requires maintaining a stack of frames containing local variables.

### Example Frame Structure

```
struct Frame {

    std::unordered_map<std::string, Value> locals;

};
```

Each function invocation creates a new frame.

## 10.3.5 Module and Import System

As programs grow larger, code reuse becomes important.
Example module usage:

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

A module system allows programs to load external files and reuse definitions.

## 10.4 Advanced Language Engineering Ideas

If you continue developing the interpreter, several advanced techniques can be explored.

- **Garbage Collection**

  Instead of strict ownership with `std::unique_ptr`, a garbage collector can manage object lifetimes dynamically.

- **Bytecode Virtual Machine**

  The AST interpreter can be replaced or augmented with a bytecode-based virtual machine.

- **JIT Compilation**

  Frequently executed code can be compiled dynamically using frameworks such as LLVM.

- **Debugging Tools**

  Advanced interpreters include breakpoints, tracing, and AST inspection tools.

## 10.5 Further Learning

Building this interpreter provides a strong introduction to language implementation. To deepen your knowledge, consider studying:

- compiler construction
- virtual machine architecture
- parsing algorithms
- programming language theory

Recommended reading:

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

## 10.6 Closing Thoughts

Designing and implementing an interpreter from scratch is one of the most rewarding projects in software engineering.
It develops skills in:

- language design
- recursive algorithms
- memory management
- modern C++ programming
- software architecture

Using only the C++ standard library and a clean procedural design, you have created a functioning programming language runtime.

From here the possibilities are almost limitless. You can extend the language, experiment with new runtime designs, or explore deeper areas of compiler and virtual machine engineering.

The journey from programmer to language designer often begins with a project exactly like this one.

# Appendices

The appendices provide additional implementation details, extended examples, and practical references that support the interpreter developed in this guide.
While the main chapters focus on the conceptual and architectural aspects of building the interpreter, the appendices offer reusable code snippets, build configurations, debugging utilities, and optional extensions.
These sections are intended to help readers experiment further and extend the interpreter beyond the minimal implementation presented in the core chapters.

## Appendix A: Full Token and Tokenizer System

### 10.6.1 Token Types

### Update File: `tokens.hpp`

```
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

### Update File: `tokens.hpp`

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

Each token stores the lexeme together with its location in the source code. Tracking line and column information enables precise error reporting during parsing and evaluation.

### 10.6.3 Tokenizer Extensions

### Update File: `lexer.cpp`

Example implementation for recognizing string literals.

```
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;
}
```

# Appendix B: CMake Enhancements for Multi-File Projects

## 10.6.4 Modular CMake Structure

### Update File: `CMakeLists.txt`

```
add_library(lexer lexer.cpp lexer.hpp)
```

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

This structure allows each component to evolve independently.

### 10.6.5 Unit Testing Target

### Update File: `CMakeLists.txt`

```
enable_testing()

add_executable(test_runner
    tests/test_main.cpp
)

add_test(
    NAME LexerTest
    COMMAND test_runner
)
```

Automated testing is essential for ensuring the stability of the interpreter as new features are introduced.

# Appendix C: Sample Unit Tests Using `Catch2`

## Create File: `tests/test_main.cpp`

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

TEST_CASE("Lex simple expression")
{
    std::string src = "x = 42 + 3";

    Lexer lexer(src);
    auto tokens = lexer.tokenize();

    REQUIRE(tokens.size() == 5);
    REQUIRE(tokens[0].type == TokenType::Identifier);
    REQUIRE(tokens[2].type == TokenType::Number);
}
```

# Appendix D: Template for Expression Evaluation

## Update File: `interpreter.cpp`

```
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 variable | Variable used before assignment | Verify that the variable exists in the symbol table |
| Token mismatch | Unexpected character in input | Extend tokenizer logic to recognize additional symbols |
| Empty AST | Parser failed to construct nodes | Ensure grammar rules cover all valid input cases |
| Infinite loop | Loop condition never becomes false | Use debug output to inspect runtime values |
| Missing tokens | Lexer skipped characters | Verify punctuation and operator handling |

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

### 10.6.6 Function Node

### Update File: `ast.hpp`

```
struct Function {

    std::string name;

    std::vector<std::string> params;

    std::vector<Stmt> body;
};
```

### 10.6.7 Function Call Node

### Update File: `ast.hpp`

```
struct Call {

    std::string funcName;

    std::vector<Expr> arguments;
};
```

### 10.6.8 Evaluator Extension

### Update File: `interpreter.cpp`

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

Value eval_call(const Call& call)
{
    auto& func =
        functions[call.funcName];

    // create local environment
    // bind parameters
    // execute body
}
```

## Appendix G: Extended Grammar Example

```
program        → statement*

statement      → expr_stmt
               | if_stmt
               | while_stmt
               | func_decl

expr_stmt      → IDENT "=" 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 the Interpreter

## 10.6.9 Building with CMake

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

## 10.6.10 Running the Interpreter

```
./interpreter_cli
```

## 10.6.11 Sample REPL Session

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

# Appendix I: REPL Loop with Error Handling

```
int main()
{
    while (true)
    {
        std::cout << ">>> ";

        std::string input;

        if (!std::getline(std::cin, input))
            break;

        try {

            Lexer lexer(input);
            auto tokens = lexer.tokenize();

            Parser parser(tokens);
            auto ast = parser.parse();

            for (const auto& stmt : ast)
                execute_stmt(stmt);
        }
        catch (const std::exception& e) {

            std::cerr
                << "Error: "
                << e.what()
                << "\n";
        }
    }
```

```
    return 0;
}
```

# Appendix J: Minimal AST Visualization

## Create File: `ast_debug.hpp`

```
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);
        }

    }, expr);
}
```

# Appendix K: Complete Project Source Tree

```
interpreter/

  CMakeLists.txt
  main.cpp

  tokens.hpp

  lexer.hpp
  lexer.cpp

  parser.hpp
  parser.cpp

  ast.hpp

  interpreter.hpp
  interpreter.cpp
```

```
repl.hpp
repl.cpp

utils.hpp
errors.hpp

ast_debug.hpp

tests/
    test_main.cpp
```

This modular structure mirrors the architecture used in many real programming language runtimes and provides a solid foundation for further experimentation and language extensions.

# References

This booklet emphasizes recent and modern references. The following materials were selected to ensure that readers are guided by sources published in or after 2020, with particular focus on Modern C++, interpreter construction, compiler engineering, and contemporary CMake practice.
These references provide both theoretical foundations and practical engineering guidance for building programming language tools using Modern C++.

## Modern C++ Standards and Core References

1. **ISO/IEC 14882:2024(E)** — *Programming Language C++*

    - The official ISO publication representing the C++23 standard.
    - This standard defines the language and library features used throughout this booklet, including `std::variant`, `std::visit`, `std::optional`, `std::unique_ptr`, and `std::string_view`.

2. **cppreference.com**

    - A continuously maintained technical reference widely used by modern C++ developers.

- Particularly useful for verifying the behavior of modern standard library components and language features.

3. **C++ Core Guidelines** — Standard C++ Foundation

   - A widely respected set of best practices for writing safe and maintainable C++.
   - Especially useful for ownership rules, RAII, and interface design when building interpreter components.

# Interpreter and Compiler Construction

1. **Programming Language Pragmatics** — Michael L. Scott and Jonathan Aldrich, 5th Edition, 2025

   - A comprehensive academic reference on programming language design and implementation.
   - Provides detailed explanations of grammars, runtime models, type systems, and language semantics.

2. **Engineering a Compiler** — Keith D. Cooper and Linda Torczon, 3rd Edition, 2022

   - A modern reference on compiler architecture and implementation strategies.
   - Useful for understanding parsing architecture, AST design, semantic analysis, and the evolution from interpreters to full compiler pipelines.

# Modern C++ Design and Engineering

1. **C++ Software Design** — Klaus Iglberger, 2022

    - A modern guide to software architecture using contemporary C++.
    - Useful for structuring interpreter components such as the lexer, parser, runtime environment, and REPL.

2. **Software Design with C++** — Klaus Iglberger, 2023

    - Expands on modern C++ design principles and component architecture.
    - Particularly helpful when scaling a small interpreter into a modular language system.

3. **Embracing Modern C++ Safely** — John Lakos, Vittorio Romeo, Rostislav Khlebnikov, and Alisdair Meredith, 2021

    - A practical guide to using modern C++ features responsibly in large software systems.
    - Relevant for ownership models, safe abstractions, and long-term maintainability.

4. **C++ Move Semantics: The Complete Guide** — Nicolai M. Josuttis, 2020

    - A detailed exploration of move semantics and ownership transfer.
    - Important for understanding efficient manipulation of `std::unique_ptr` and other move-only types used in AST structures.

# CMake and Development Tooling

1. **Professional CMake: A Practical Guide** — Craig Scott

    - One of the most authoritative resources on modern CMake usage.
    - Provides practical guidance for building modular multi-file projects, managing dependencies, and creating test targets.

2. **Official CMake Documentation**

    - The primary reference for understanding modern target-based CMake design.
    - Essential for managing build configuration and cross-platform compilation.

3. **Catch2 Documentation**

    - Documentation for the Catch2 testing framework used for lightweight unit testing.
    - Useful for validating lexer, parser, interpreter, and REPL behavior.

# Practical Development Resources

1. **Compiler Explorer (Godbolt)**

    - A valuable online tool for inspecting compiler output and experimenting with Modern C++ features.
    - Useful for validating language behavior and comparing compiler implementations.

2. **Educational Open-Source Interpreter Projects**

- Recent interpreter and small-language implementations written in Modern C++ provide useful practical examples.
- These projects can be helpful for comparing AST design, parser architecture, runtime environments, and testing approaches.

## Recommended Study Strategy

Readers who wish to deepen their understanding of interpreter construction should consider the following study approach:

- Use the **C++23 standard** and **cppreference** for precise language and library behavior.
- Use **Programming Language Pragmatics** for theoretical background on language semantics and runtime systems.
- Use **Engineering a Compiler** for deeper understanding of parser design and compiler architecture.
- Use **Professional CMake** to structure larger interpreter projects and manage build systems effectively.
- Use modern C++ design literature to improve the safety, clarity, and maintainability of interpreter implementations.