Notes on Crafting Interpreters by Robert Nystrom

Jason Mac

August 28, 2025

Contents

1 Parts of a Language and the Lox Language

1.1 Overview

Credit: Robert Nystrom, *Crafting Interpreters*. This set of notes is directly sourced from this book as linked. It is an extremely well crafted text that gives a concise, yet extremely insightful introduction to creating your own progmamming language.

A language can be broken up into many parts. This set of notes taken from Crafting Interpreters is concerned about its impementation. In particular, we will be creating an interpreter for the Lox language as created by Robert Nystrom using C#. A language's impementation can be pieced down into separate parts. It's useful to make the abstraction to create intermediate representation of the source code to remain indendent of the system we are compiling or interpreting on. The remainder of section will give an overview on the Lox langauge and its specifications.

1.2 Hello, Lox

Your first Lox program:

```
// Print "Hello, world!" in Lox
print "Hello, world!";
```

Lox has C-family syntax: semicolons end statements, 'print' is a built-in statement, and parentheses are generally not required around arguments.

1.3 A High-Level Language

Lox is small, high-level, and dynamically typed. Its design is inspired by languages like JavaScript, Scheme, and Lua:

1.3.1 Dynamic Typing

Variables can store values of any type and change type at runtime. Type errors are detected only when code executes. The author states that, although static typing is useful in many scenarios, it is more trouble that it's worth to implement it for the core learning purposes.

1.3.2 Automatic Memory Management

Memory is handled automatically via garbage collection. Lox avoids manual allocation and deallocation, simplifying programming. The implemntation of the garbage collector will be done by us.

1.4 Data Types

Lox supports a small set of built-in types:

- **Booleans**: `true`, `false`
- **Numbers**: double-precision floating point, e.g., `1234;` or `12.34;`
- **Strings**: enclosed in double quotes, e.g., `"Hello"`

- **Nil**: represents absence of a value (`nil`)

It is stated that it is more trouble than its worth to ban the `nil` value from the language since it is dynamically typed. It is interesting to note that the introduction of the `null` value in C is considered the **"billion-dollar mistake"** by Tony Hoare, its own inventor.

1.5 Expressions

Expressions combine literals and operators to produce values. If built-in data types and their literals are atoms, then expressions must be the molecules. For example, an arithmetic expression in Lox could be:

```
1 + 2 * 3
```

Here, "1", "2", and "3" are literals, and "+" and "*" are operators combining them into an expression.

1.5.1 Arithmetic

Lox has the basic arithmetic operators that are common to many languages, perhaps necessary to be turing-complete.

```
add + me;
subtract - me;
multiply * me;
divide / me;
-negateMe;
```

Operands and Operators:

- **Operand:** A value that an operator acts on. Example: in `3 + 4`, both `3` and `4` are operands.
- **Binary operator:** An operator with two operands. Example: `+`, `-`, `*`, `/`.
- **Infix operator:** Operator appears **between** operands. Example: `3 + 4`.
- **Prefix operator:** Operator appears **before** its operand. Example: `-5`.
- **Postfix operator:** Operator appears **after** its operand. Example: `i++` in C.

1.5.2 Comparison and Equality

Lox includes comparison operators that will produce a true or false value.

```
less < than;
lessThan <= orEqual;
greater > than;
greaterThan >= orEqual;
```

Equality and Inequality:

- Compare any values with `==` (equals) or `!=` (not equals).

- Examples:
  - `1 == 2` → false
  - `"cat" != "dog"` → true
  - `314 == "pi"` → false
  - `123 == "123"` → false
- Values of different types are never equivalent; no implicit conversions.

1.5.3 Logical Operators

Lox provides negation of booleans through the prefix operator `!` and works as expected. The conjunction and disjunction of statements is done with binary operators, `and`, `or`. These are control flow statements.
In particular, `and` returns its left operand if its false, otherwise it returns its right operand. Similarly, `or` returns its left operand if its true, otherwise it returns its right operand. The operator is a short circuit.

```
!true;          // false
true and false; // false
true or false;  // true
```

1.6 Statements

Statements: Produce effects rather than values (unlike expressions). Statments, as metioned by Nystrom, by definition, don't evaluate to value. To be useful, they must either modify some state, read input, or produce output.
- `print "Hello, world!";` → evaluates expression and displays the result (produce output)
- Expression statement: an expression followed by `;` is promoted to a statement.
Example: `"some expression";`
- **Blocks:** Wrap multiple statements in `{ ... }`. Blocks affect scoping.

```
{
  print "One statement.";
  print "Two statements.";
}
```

1.7 Variables

Declared with `var`. Uninitialized variables default to `nil`. When a variable has been declared you can begin to access its value and assign a varible through its name.

```
var breakfast = "bagels";
print breakfast; // "bagels"
breakfast = "beignets";
print breakfast; // "beignets"
```

1.8 Control Flow

Lox allows for control statements that are analogous to those seen in C.

- **If statement:**

  ```
  if (condition) {
    print "yes";
  } else {
    print "no";
  }
  ```

- **While loop:**

  ```
  var a = 1;
  while (a < 10) {
    print a;
    a = a + 1;
  }
  ```

- **For loop:**

  ```
  for (var a = 1; a < 10; a = a + 1) {
    print a;
  }
  ```

In other languagues you might be familiar with the `for-in` or `foreach` loop. Lox will not provide those functionalities. Although in design specification, this could be a nice to have.

1.9 Functions

Functions are declared with `fun` and are similar to those calls seen in C:

- Call a function:

  ```
  makeBreakfast(bacon, eggs, toast);
  makeBreakfast(); // no arguments
  ```

  Parentheses are mandatory; omitting them refers to the function itself.

- Define a function using `fun`:

  ```
  fun printSum(a, b) {
      print a + b;
  }
  ```

- **Terminology: Argument vs Parameter**:
  - **Argument**: actual value passed when calling a function.

  - We say that a function *call* has a **argument list** or actual paramter.
  - **Parameter**: variable in function definition that holds the argument's value.
  - We say that a function *declaration* has a **parameter list** or formal parameters/formals.

- In Lox (dynamically typed), function declarations and definitions are the same.
- Function body is always a block. Use `return` to produce a value:

  ```
  fun returnSum(a, b) {
      return a + b;
  }
  ```

  If no return is reached, function implicitly returns `nil`.

1.9.1 Closures

Functions are first-class values in Lox.

Definition 1.1: First-Class Function

A ***first-class function*** is a function that can be treated like any other value in the language. Specifically, a first-class function can be:

- Assigned to variables
- Passed as arguments to other functions
- Returned from other functions
- Stored in data structures

Languages in which functions are first-class allow higher-order programming, enabling functions to be used as building blocks for more abstract operations.

Example:

```
fun addPair(a, b) { return a + b; }
fun identity(a) { return a; }


print identity(addPair)(1, 2); // 3
```

In this example, we are passing in `addPair` into `identity` which just returns the function itself. It then consumes the argument list `(1, 2)` once returned.

Local Functions: Functions can be declared inside other functions.

```
fun outerFunction() {
  fun localFunction() {
    print "I'm local!";
  }
```

```
  localFunction();
}
```

Definition 1.2: Closure

A ***closure*** is an ordered pair (f, E) where:

1. f is a function body, and
2. E is the ***environment*** of all variables that f references but are not parameters.

A closure allows the function f to be invoked with its captured environment E even after the scope in which it was defined has ended. Formally, for a closure (f, E) and arguments $x_1, \ldots, x_n$,

$$(f, E)(x_1, \ldots, x_n) \equiv f(x_1, \ldots, x_n) \text{ evaluated in the environment } E.$$

More simply, a **closure** is a function that captures variables from its surrounding scope. This allows the function to access those variables even after the outer function has returned.

Example:

```
fun returnFunction() {
  var outside = "outside";
  fun inner() { print outside; }
  return inner;
}

var fn = returnFunction();
fn(); // prints "outside"
```

Key Idea: Implementing closures requires bundling the function's code with the surrounding variables it references. This allows proper access to variables even after their original scope has ended.

1.10 Classes

Lox will also be an object-oriented languague. Classes in Lox can have methods, fields, inheritance, and initializers. object-oriented languagues can be controversial in its use cases. Nystrom makes the argument that since Lox is dynamically typed, it will come in handy to define compound data types and bundle data together. It is also useful since we would not have to prefix specific object methods with its specific type to avoid name collision.

Classes vs Prototypes

In object-oriented programming, there are two primary approaches to defining and organizing objects: **class-based** and **prototype-based** systems.

Definition 1.3: Class-Based Object System

A ***class-based system*** consists of ***classes*** and ***instances***:

- **Class**: A blueprint that defines the methods and inheritance relationships for its instances.
- **Instance**: An object that stores state and has a reference to its class. Method calls on an instance are resolved by looking up the method in the instance's class.

Method lookup may be:

- ***Static dispatch***: Resolved at compile-time based on the declared type of the instance (common in statically typed languages like C++ or Java).
- ***Dynamic dispatch***: Resolved at runtime based on the actual object instance (common in dynamically typed languages like Lox).

Definition 1.4: Prototype-Based Object System

A ***prototype-based system*** contains only objects; there are no classes. Each object may:

- Store its own state and methods.
- Inherit from another object ("delegates" to another object).

This approach allows for flexible object composition but can be used to simulate class-like structures.

Observation From Nystrom: Although prototype-based languages are simpler to implement, programmers often use them to recreate class-like patterns. Therefore, class-based systems remain popular for organizing object-oriented code, as in Lox.

Classes in Lox

In Lox, you can declare a class and its methods as such:

```
class Breakfast {
  cook() {
    print "Eggs a-fryin'!";
  }

  serve(who) {
    print "Enjoy your breakfast, " + who + ".";
  }
}
```

The body of a class contains its methods. Just like function declarations except *without* the `fun` keyword.

When the class declaration is executed, Lox creates a class object and stores it in a variable named after the class.

Classes are *first-class* 1.9.1 values in Lox.
Which allows you to do the following with the afformentioned `Breakfast` class:

- You can store classes in variables

  ```
  var someVariable = Breakfast;
  ```

- Pass them to functions:

  ```
  someFunction(Breakfast);
  ```

To create instances, the class itself acts as a factory function. Call a class like a function to produce a new instance:

```
var breakfast = Breakfast();
print breakfast; // "Breakfast instance"
```

Instantiation and Initialization

In object-oriented programming, encapsulating *state* and *behavior* together requires fields and methods. In Lox, fields can be added freely to objects:

```
// Adding fields dynamically
breakfast.meat = "sausage";
breakfast.bread = "sourdough";
```

Accessing fields or methods from within a method of the class uses `this`:

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

To ensure an object starts in a valid state, define an **initializer** method `init()`. It is automatically called when the class is instantiated:

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

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

```
}

// Creating an instance
var baconAndToast = Breakfast("bacon", "toast");
baconAndToast.serve("Dear Reader");
// Prints: "Enjoy your bacon and toast, Dear Reader."
```

Inheritance in Lox

Inheritance allows methods and fields to be reused across multiple classes. Lox supports *single inheritance*. When declaring a class, you can specify its parent class (superclass) using the less-than symbol `<`.

```
class Brunch < Breakfast {
  drink() {
    print "How about a Bloody Mary?";
  }
}
```

Here:

- `Brunch` is the **derived class** (or **subclass**).
- `Breakfast` is the **base class** (or **superclass**).

Formal Definitions

- **Superclass / Base class:** A class that provides methods and fields which may be inherited by another class. Formally, if class B extends class A, then A is the superclass of B.
- **Subclass / Derived class:** A class that inherits methods and fields from another class, and may define additional behavior. Formally, if class B extends class A, then B is the subclass of A.

Set-Theoretic View

If $S(A)$ denotes the set of all instances of class A, and $S(B)$ denotes the set of all instances of class B, then:

$$B < A \quad \Longrightarrow \quad S(B) \subseteq S(A).$$

That is, every instance of the subclass is also an instance of the superclass.

Constructors and `super`

Subclasses may override the `init()` method (initializer). To ensure that the superclass state is properly initialized, the subclass calls the superclass's initializer using `super`.

```
class Brunch < Breakfast {
  init(meat, bread, drink) {
    super.init(meat, bread);
    this.drink = drink;
  }
}
```

1.11 Standard Library

The minimal Lox standard library includes:

```
print "Hello";
clock(); // returns time since program start
```

2 Scanning

The first step of crafting an interpreter involves **scanning** or **lexing**. In other contexts it can also be called **lexical analysis**.

Definition 2.1: Token

A **token** is a syntactic unit of a programming language, representing a categorized sequence of characters in the source code. Each token consists of:

- **Type:** the category of the token (e.g., `IDENTIFIER`, `NUMBER`, `KEYWORD`).
- **Lexeme:** the exact sequence of characters in the source.
- **Literal (optional):** the value represented by the token (e.g., numeric value or `null`).

Tokens are produced by the **lexer** and serve as atomic units for parsing.

Definition 2.2: Scanner

A **Scanner** is a processor that takes in an input of linear stream of characters to produce a collection of tokens.

This can also be called a **lexer**.
For example, suppose we have the following stream of characters:

$$var\, avergage = (min + max)/2;$$

In the definition, of our language we would want want the scanner to piece this into the following collection of tokens:

$$\{[var], [average], [=], [(,][+], [min], [+], [max], [/], [2], [;]\},$$

For any possible source code we are given, the first step would to create this collection of tokens from source.

3 Representing Grammar

- Source code is first transformed into **tokens** 2.

- Tokens are then transformed into a richer representation:
  - Simple for the parser to produce.
  - Easy for the interpreter to consume.
- This representation serves as the bridge between parsing and interpretation.
- The section is concerned with formal grammars, design patterns, and programming paradigms to define it.

Since arithmetic expressions follow an order of operations, such as multiplication before addition or subtraction, they can be visualized using trees. In these trees, the leaf nodes represent numbers, while the interior nodes represent operators with branches for their operands. Evaluation proceeds from the leaves up to the root in a post-order traversal: sub-expressions are evaluated first, and then their results are combined step by step until the final value is obtained.

3.1 Context Free Grammars

Previously, the rules governing the lexical grammar, the way we grouped characters into tokens, were expressed using regular languages. This was sufficient for scanning because the scanner only needs to produce a flat sequence of tokens. However, regular languages lack the expressive power required for describing nested structures, such as arithmetic expressions which nest arbitrarily deeply. To capture these, we require a more powerful formalism: the context-free grammar (CFG).

3.1.1 Formal Grammars

A *formal grammar* begins with an *alphabet*, a finite set of atomic symbols. From this alphabet, it defines a (potentially infinite) set of valid *strings*, where each string is a finite sequence of symbols from the alphabet. The grammar provides rules that specify exactly which strings are well-formed or valid.

Definition 3.1: Context-Free Grammar

A **context-free grammar (CFG)** is a 4-tuple:

$$G = (V, \Sigma, R, S)$$

where:

- V is a finite set of ***nonterminal symbols***, which act as variables that can be expanded.
- Σ is a finite set of ***terminal symbols***, disjoint from V, representing the basic symbols of the language (e.g., tokens).
- R is a finite set of ***production rules*** of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.
- $S \in V$ is the ***start symbol***, from which derivations begin.

A string $w \in \Sigma^*$ is in the language of G if $S \Rightarrow^* w$, meaning w can be derived from the start symbol by repeatedly applying production rules.

3.1.2 Lexical vs. Syntactic Grammars

In lexical grammars, the alphabet is made up of characters, and the valid strings correspond to lexemes (that is, tokens). In syntactic grammars, by contrast, the alphabet consists of tokens, and the valid strings are entire expressions. Thus, moving from lexical analysis to parsing means working at a higher level of granularity.

Terminology	**Lexical Grammar**	**Syntactic Grammar**
The “alphabet” is	Characters	Tokens
A “string” is	Lexeme or token	Expression
It is implemented by	Scanner	Parser

Example

The purpose of a formal grammar is to distinguish valid strings from invalid ones. For example, in a grammar of English sentences, the sequence “eggs are tasty for breakfast” would be valid, whereas “tasty breakfast for are eggs” would not. Similarly, in programming languages, a syntactic grammar determines which token sequences form well-structured programs and which do not.

Lexical vs. Syntactic Grammars

In lexical grammars, the alphabet is made up of characters, and the valid strings correspond to lexemes (that is, tokens). In syntactic grammars, by contrast, the alphabet consists of tokens, and the valid strings are entire expressions. Thus, moving from lexical analysis to parsing means working at a higher level of granularity.

3.1.3 Rules for Grammars

A grammar defines an infinite set of valid strings using a finite number of *production rules*. Each rule specifies how symbols can be replaced, and strings derived from these rules are called *derivations*. Because they generate valid strings, rules are also called *productions*.

Definition 3.2: Production Rule

In a context-free grammar, a ***production rule*** has the form

$$A \to \alpha$$

where:

- A is a single ***nonterminal symbol***.
- $\alpha \in (V \cup \Sigma)^*$ is a sequence of terminals and nonterminals.

Here, V is the set of nonterminals and Σ the set of terminals.

Definition 3.3: Terminal and Non-Terminal

- A ***terminal*** is a symbol from the grammar's alphabet that cannot be replaced further. In programming languages, these correspond to tokens such as keywords or literals.
- A ***nonterminal*** is a symbol that refers to another rule in the grammar. Expanding a nonterminal means applying one of its production rules.

- Grammars define infinite sets of strings by recursive application of finite rules.
- A single nonterminal may have multiple productions, allowing alternative expansions.
- Terminals represent atomic tokens; nonterminals encode grammatical structure.
- Recursion in productions is what allows grammars to describe infinite, nested, or balanced structures, which regular grammars cannot capture.

Example: Breakfast Grammar Using a simple notation inspired by Backus–Naur form (BNF), we define a grammar for breakfast menus:

breakfast	→ protein "with" breakfast "on the side" | protein | bread
protein	→ crispiness "crispy" "bacon" | "sausage" | cooked "eggs"
crispiness	→ "really" | "really" crispiness
cooked	→ "scrambled" | "poached" | "fried"
bread	→ "toast" | "biscuits" | "English muffin"

Generating a String

Starting with the nonterminal breakfast, one possible derivation is:

$$\begin{aligned}
\text{breakfast} &\Rightarrow \text{protein "with" breakfast "on the side"}\\
&\Rightarrow \text{cooked "eggs" "with" breakfast "on the side"}\\
&\Rightarrow \text{"poached" "eggs" "with" breakfast "on the side"}\\
&\Rightarrow \text{"poached" "eggs" "with" bread "on the side"}\\
&\Rightarrow \text{"poached" "eggs" "with" "English muffin" "on the side"}
\end{aligned}$$

Thus the generated string is:

"poached eggs with English muffin on the side"

This illustrates how recursion, multiple productions, and the distinction between terminals and nonterminals allow grammars to encode infinitely many valid sentences with finite rules.

3.1.4 Enhanced Grammar Notation

To simplify the expression of grammars, we introduce several syntactic shortcuts. Beyond terminals and nonterminals, we allow:

- **Alternatives** using the pipe symbol (`|`), letting a rule list multiple possible productions.
- **Grouping** with parentheses to apply operations to a group of symbols.
- **Postfix operators**: Applies to the production proceeding it.
  - `*` — zero or more repetitions.
  - `+` — one or more repetitions.
  - `?` — optional (zero or one occurrence).

Using these enhancements, the breakfast grammar can be written as:

$$\begin{array}{ll}
\text{breakfast} & \rightarrow \text{protein ("with" breakfast "on the side")? } \mid \text{ bread}\\
\text{protein} & \rightarrow \text{"really"+ "crispy" "bacon"}\\
 & \mid \text{ "sausage"}\\
 & \mid \text{ ("scrambled" } \mid \text{ "poached" } \mid \text{ "fried") "eggs"}\\
\text{bread} & \rightarrow \text{"toast" } \mid \text{ "biscuits" } \mid \text{ "English muffin"}
\end{array}$$

This notation allows us to express optional elements, repetitions, and multiple alternatives succinctly without creating extra rules. It resembles EBNF or regular-expression style syntax but operates on token sequences rather than individual characters. These rules will inform the parse trees that represent code in memory and guide our parser's implementation.

3.1.5 A Grammar for Lox Expressions

Previously we had constructed the entirety of Lox's lexical grammar at one go. Given that the syntactic grammar is larger, it is imperative to construct it bit by bit.

Thus, we will concern ourselves with particular subsets of the language before moving onto more complex portions of the grammar.

To start, we will begin implementation of the following:

- **Literals**: Numbers, strings, Booleans, and nil.
- **Unary Expressions**: `!` for logical not and `-` to negate a number.
- **Binary Expressions**: Infix `+, -, *, /` and logical operators `==, !=, <, <=, >, >=`.
- **Parantheses**: A pair, `(` and `)` wrapped around an expression.

Doing so, allows for syntax expressions like: `1 - (2*3) < 4 == false`.

At a first attempt, we construct the following:

```
expression   → literal
             | unary
             | binary
             | grouping ;

literal      → NUMBER | STRING | "true" | "false" | "nil" ;
grouping     → "(" expression ")" ;
unary        → ("-" | "!") expression ;
binary       → expression operator expression ;
operator     → "==" | "!=" | "<" | "<=" | ">" | ">="
             | "+" | "-" | "*" | "/" ;
```

3.2 Implementing Syntax Trees

The expression grammar we defined is recursive, which naturally forms a **tree structure** representing the syntax of the language. This structure is called an *Abstract Syntax Tree (AST)*. Unlike a parse tree, an AST omits unnecessary grammar productions that are not needed for later stages of interpretation or compilation.

Expressions in an AST are **heterogeneous**:

- **Unary expressions** have one operand.
- **Binary expressions** have two operands.
- **Literals** have no operands.

To represent this in code, it is common to define a **base class** for all expressions (e.g., `Expr`) and create **subclasses** for each expression type (e.g., `Binary`, `Unary`, `Literal`). This approach enforces **type safety** by allowing the compiler to catch invalid field accesses.

Tokens in the AST are also not completely homogeneous: literals store values, while other token types may not require additional state.

Nesting the subclasses inside the base class is a practical organizational choice, keeping related expression types together in a single file without affecting functionality.

Observation: This design emphasizes **clarity and type safety** over compactness, making it easier to reason about the tree structure in a statically typed language like Java.

4 Parsing

The **parsing** step involves giving the syntax of our language a **grammar**, which would allow us to compose larger expressions and statements out of atomic units.

Definition 4.1: Parser

A **parser** is a program or algorithm that takes as input a sequence of tokens (produced by a scanner/lexer) and constructs a tree-like representation of the input according to the rules of a formal grammar.
The resulting structure is called a **parse tree** or **abstract syntax tree (AST)**, depending on whether it retains all syntactic details or abstracts some of them.
Formally, given a grammar $G = (V, \Sigma, R, S)$, where:

- V = set of non-terminal symbols,
- Σ = set of terminal symbols,
- R = set of production rules,
- S = start symbol,

a parser is a function

$$\text{parse} : \Sigma^* \to \text{Tree}(V \cup \Sigma)$$

such that for a valid token sequence $w \in \Sigma^*$, $\text{parse}(w)$ produces a tree T that represents a derivation of w from the start symbol S using the rules R.
The parser also detects and reports syntax errors when the input does not conform to the grammar.

4.1 Ambiguuity and the Parsing Game

In a grammar, there can be more than one way to achieve a value $\omega \in \Sigma^*$.

E.g. consider the grammer we constucted previously along with sequence of strings and its collection of tokens:

$6/3 - 1 = (6/3) - 1$
$\{[6], [/], [3], [-], [1]\}$

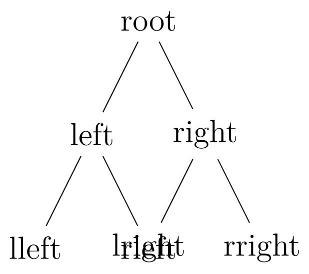
```
expression   → literal
             | unary
             | binary
             | grouping ;

literal      → NUMBER | STRING | "true" | "false" | "nil" ;
grouping     → "(" expression ")" ;
unary        → ( "-" | "!" ) expression ;
binary       → expression operator expression ;
operator     → "==" | "!=" | "<" | "<=" | ">" | ">="
             | "+" | "-" | "*" | "/" ;
```

To construct such a sequence given the grammar we can do both:

- $$\begin{aligned} Expression &\to binary_1 \tag{1}\\ binary_1 &\to NUMBER(6)\ operator(/)\ binary_2 \tag{2}\\ binary_2 &\to NUMBER(3)\ operator(-)\ NUMBER(1) \tag{3}\end{aligned}$$

- $$\begin{aligned} Expression &\to binary_1 \tag{1}\\ binary_2 &\to NUMBER(3)\ operator(-)\ NUMBER(1) \tag{2}\\ binary_1 &\to NUMBER(6)\ operator(/)\ binary_2 \tag{3}\\ & \tag{4}\end{aligned}$$

In this instance, our language has a "valid" string except that there's more that one way to achieve it, hence they will not produce the same syntax trees upon construction.

too much work lol 

Then upon evaluation, they will produce not the same values. This causes an issues thus we must define a set of precedence with out operators.

To maintain semantics of desired functionality, we define the following:

Definition 4.2: Precedence

Precedence determines which operator is evaluated first in an expression containing a mixture of different operators. Operators with higher precedence are evaluated before operators with lower precedence. Operators with higher precedence are said to ***bind tigher***.

Definition 4.3: Associativity

Associativity determines which operator is evaluated first in an expression in a series of the *same* opeartor.

- When an operator is ***left-associative***, operators on the left are evaluated first before those on the right
- When an operator is ***right-associative***, operators on the right are evaluated first before those on the right
- When an operator is ***non-associative*** this means that it's an error to use that operator more than once in a sequence.
- $\exists$ languages s.t. $\exists$ certain operators have no precedence.

E.g. For left-associative, $5 - 3 - 1 \equiv (5 - 3) - 1$

E.g. For right-associative, $a = b = b \equiv a = (b = c)$.

When writing source code, the writer would not need to explicity state the groupings as the syntax tree should handle that itself to maintain semantic integrity.

For lox, we will use the following precedence form lowest to highest as in C:

Name	**Operators**	**Assoicates**
Equality	== !=	Left
Comparison	> >= < <=	Left
Term	- +	Left
Factor	/ *	Left
Unary	! -	Right

The current construction allows for the grammar to accept expressions regardless of the defined precedence, to fix this we define in the grammar a separate rule for each precedence level.

Hence, we have the following grammar rules that forces the grammar to adhere to our semantics: Primary rule holds all literals and parenthesized expressions which are at the top of precedence. For adherance, we enforce that each production rule can only match expressions at its precedence level or higher.

```
expression   → equality
equality →   comparison ( ("==" | "!=" ) comparison )* ;
comparison   → term ( (">" | ">=" | "<" | "<=") term )* ;
term         → factor ( ("-" | "+") factor )* ;
factor       → unary ( ("/" | "*") unary )* ;
unary        → ("!" | "-") unary
             | primary
primary      → NUMBER | STRING | "true" | "false" | "nil"
             | "(" expression ")" ;
```

4.2 Recurive Descent Parsing

Definition 4.4: Rersursive Descent Parsing

Recurive Descent Parsing is a **top-down parser**. It starts from the outermost or top grammar rule, (expreession) and works its way down nested subexpressions and bottoms out at the leaves of the syntax tree.

In essence, rdp is a literaly translation of the grammar into imperative code. Each rule is a function associated with its implementation determined by the precedence. It is called "recursive" since grammar rules can refer to itself leading to recursive calls.