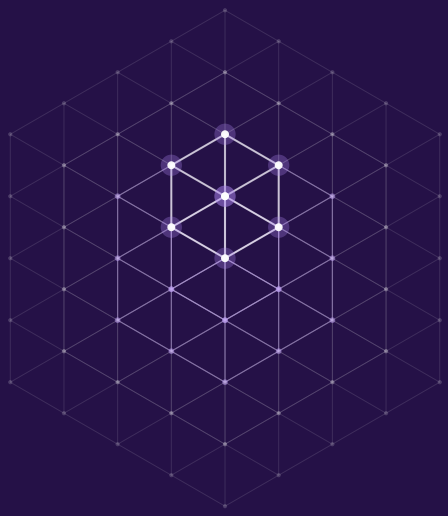
The Lattice Handbook

A Comprehensive Guide to the Lattice Programming Language



Alex Jokela

Covers Lattice v0.3.28

The Lattice Handbook
A Comprehensive Guide to the Lattice Programming Language

ISBN 979-8-9951255-1-8

Covers Lattice v0.3.28.

First edition, 2026.

Contents

Part I

First Contact

Chapter 1

Hello, Lattice

Every programming language begins with a promise. Some promise speed. Others promise safety, or simplicity, or raw expressive power. Lattice promises something different: it promises you *control over change itself*.

In most languages, the question of whether a value can be modified—or when it should stop being modified—is answered by convention, or by a single keyword like `const`. Lattice takes that question seriously and builds an entire system around it, drawing on a metaphor from chemistry and materials science. Values in Lattice are like materials: they can be *fluid*, flowing and reshaping freely, or they can *crystallize* into a permanent, immutable form. This is the *phase system*, and it is woven into every corner of the language.

But we are getting ahead of ourselves. Before we explore crystallization, let's meet Lattice the way any programmer meets a new language—by writing our first program and watching it run.

1.1 What Lattice Is and Why It Exists

Lattice is an interpreted, general-purpose programming language implemented in C. It compiles your source code to bytecode and executes it on a stack-based virtual machine, with the option to run on a register-based VM or a tree-walking interpreter as well. The current release is v0.3.28, and it runs on macOS, Linux, and (experimentally) via WebAssembly in the browser.

On the surface, Lattice looks familiar. If you've written Python, JavaScript, Rust, or Go, you'll recognize the curly braces, the `if`/`else` blocks, and the `fn` keyword for functions:

Listing 1.1: A first glance at Lattice syntax

```
fn greet(name: String) -> String {
    "Hello, ${name}!"
}

print(greet("world"))
// Hello, world!
```

So what makes Lattice worth learning? Three things stand out.

1.1.1 The Phase System

Most languages draw a binary line: a variable is either mutable or immutable. Lattice replaces that binary with a richer model. A value starts *fluid*—you can modify it, grow it, reshape it. When you are done, you *freeze* it, and it becomes *crystal*—locked, permanent, safe to share across threads. If you need to modify it again later, you *thaw* it, which gives you a mutable copy without disturbing the original.

This is not merely a convention. The Lattice runtime enforces it. Try to mutate a frozen value and you will get an error—not a silent bug discovered in production at 2 a.m.

Listing 1.2: Freezing and thawing

```
flux temperatures = [72.1, 68.5, 74.3]
temperatures.push(71.0)   // fine: temperatures is fluid

freeze(temperatures)
// temperatures.push(80.0) -- error! crystal values cannot be modified

flux updated = thaw(temperatures)  // thaw produces a fluid copy
updated.push(80.0)                 // now we can modify again
```

We will explore the phase system gradually throughout this book, with a full deep-dive in Part III.

1.1.2 Familiar Syntax, Modern Features

Lattice ships with first-class closures, pattern matching with exhaustiveness checking, structs with traits and `impl` blocks, string interpolation, lazy iterators, structured concurrency with channels, and

a built-in test framework. It is a language designed for working programmers who want to get things done without drowning in ceremony.

1.1.3 Batteries Included

Lattice includes over 120 built-in functions spanning file I/O, HTTP, JSON/TOML/YAML parsing, regular expressions, cryptography, networking, and more. The standard library modules can be imported with a single line. There is also a package manager for third-party libraries.

Lattice Is Young

Lattice is at version 0.3.28. The language is actively evolving. Some features described in this book may change in future releases, though the core concepts—the phase system, the type model, the concurrency story—are stable foundations that the rest of the language builds upon.

1.2 Installing Lattice

Lattice is built from source using a C11 compiler and `make`. There is no pre-built binary distribution yet—part of the charm of a young language is that you get to compile it yourself.

1.2.1 Prerequisites

You need two things:

1. A **C11-compatible compiler**: GCC or Clang on Linux, Apple Clang on macOS, or MinGW on Windows.
2. **libedit**: a lightweight readline library used for the REPL's line editing and tab completion. It ships with macOS. On Linux, install the development package.

Optionally, if you want TLS networking (`tls_connect`, `tls_read`, etc.) and cryptographic built-ins (`sha256`, `md5`, `base64_encode`), you'll need **OpenSSL** available via `pkg-config`.

1.2.2 macOS

macOS ships with Apple Clang and libedit. Clone the repository and build:

```
git clone https://github.com/alexjokela/lattice.git
cd lattice
make
```

This produces a binary called `clat` in the current directory. You can move it to a directory on your `PATH`:

```
sudo cp clat /usr/local/bin/
```

1.2.3 Linux

On Debian or Ubuntu, install the libedit development headers first:

```
sudo apt install libedit-dev
git clone https://github.com/alexjokela/lattice.git
cd lattice
make
```

On Fedora or RHEL:

```
sudo dnf install libedit-devel
```

1.2.4 Windows

Lattice includes a Win32 compatibility layer (`src/win32_compat.c`) that provides POSIX-like functions. Using MinGW or WSL:

```
# Using WSL (recommended)
sudo apt install libedit-dev build-essential
cd lattice
make
```

WSL Is the Smoothest Path

If you are on Windows, the Windows Subsystem for Linux gives you the best experience. Lattice was primarily developed on macOS and Linux, and WSL brings you into that environment without dual-booting.

1.2.5 WebAssembly

Lattice can be compiled to WebAssembly using Emscripten, allowing it to run in a browser. This is experimental, but it opens up interesting possibilities for playgrounds and education:

```
# Requires Emscripten SDK installed and activated
emmake make
```

The `platform()` built-in returns `"wasm"` when running in this mode.

1.2.6 Verifying the Installation

Once you have built `clat`, verify it works:

```
./clat --version
```

You should see:

```
Lattice v0.3.28
```

If you want the full help screen with all available subcommands and flags:

```
./clat --help
```

1.3 Your First Program: Hello, World!

Let's write our first Lattice program. Create a file called `hello.lat`:

Listing 1.3: hello.lat — your first Lattice program

```
print("Hello, World!")
```

Run it:

```
./clat hello.lat
```

Output:

```
Hello, World!
```

That's it. No `main()` function required, no boilerplate, no imports. Lattice programs are sequences of statements, and the interpreter executes them top to bottom.

1.3.1 A Slightly More Interesting Program

Let's make it personal:

Listing 1.4: greeting.lat — string interpolation

```
let name = "Lattice"
let year = 2026
print("Welcome to ${name}! The year is ${year}.")
```

Output:

```
Welcome to Lattice! The year is 2026.
```

Notice the `${...}` syntax inside double-quoted strings. Any expression can go inside those braces—variable lookups, arithmetic, even function calls. We will cover string interpolation in depth in Chapter 3.

1.3.2 Functions

Here is a program with a function:

Listing 1.5: A function with a type annotation

```
fn fahrenheit_to_celsius(temp: Float) -> Float {
    (temp - 32.0) * 5.0 / 9.0
}

let boiling = fahrenheit_to_celsius(212.0)
print("Water boils at ${boiling} Celsius")
// Water boils at 100.0 Celsius
```

A few things to notice:

- Function parameters have **type annotations**: `temp: Float` declares that `temp` must be a `Float`. Pass the wrong type and Lattice will tell you—at runtime, with a clear error message.
- The **last expression** in a function body is the implicit return value. No `return` keyword needed (though you can use one if you prefer explicit style).
- The `-> Float` after the parameter list declares the return type.

1.3.3 Running the Examples Directory

Lattice ships with an `examples/` directory containing programs that demonstrate different features. Try running the Fibonacci example:

```
./clat examples/fibonacci.lat
```

This program computes Fibonacci numbers using both iterative and recursive approaches, verifies they produce the same results, and approximates the golden ratio. Here is a simplified version of what it does:

Listing 1.6: A taste of the Fibonacci example

```
fn fib_iterative(n: Int) -> Int {
    if n <= 1 { return n }
    flux a = 0
    flux b = 1
    flux i = 2
    while i <= n {
        let temp = a + b
        a = b
        b = temp
        i += 1
    }
    return b
}

for i in 0..10 {
    print("fib(${to_string(i)}) = ${to_string(fib_iterative(i))}")
}
```

Notice the `flux` keyword: it declares mutable variables. We will explain `flux`, `let`, and `fix` in Chapter 4.

1.4 Running Files vs. the REPL

There are two main ways to run Lattice code: from a file, or interactively in the REPL.

1.4.1 Running from a File

You've already seen this:

```
./clat my_program.lat
```

Lattice reads the entire file, compiles it to bytecode, and executes it on the stack VM. If you want to use the tree-walking interpreter instead (slower, but useful for debugging), pass `--tree-walk`:

```
./clat --tree-walk my_program.lat
```

You can also *pre-compile* a file to bytecode and run the compiled output later:

```
./clat compile program.lat -o program.latc
./clat program.latc
```

The `.latc` file contains serialized bytecode. Running a pre-compiled file skips the lexing, parsing, and compilation steps, which can speed up startup for larger programs.

Three Execution Backends

Lattice has three ways to execute your code: the **stack-based bytecode VM** (the default and fastest), the **register-based VM** (via `--regvm`), and the **tree-walking interpreter** (via `--tree-walk`). All three support the full language. We will cover the backends in depth in Chapter 33 (*The Three Backends*)—for now, the default is the right choice.

1.4.2 The Interactive REPL

Launch the REPL by running `clat` with no arguments:

```
./clat
```

You'll see a welcome banner:

```
Lattice v0.3.28 -- crystallization-based programming language
Copyright (c) 2026 Alex Jokela. BSD 3-Clause License.
Type expressions to evaluate. Ctrl-D to exit.
```

Now you can type expressions and see results immediately:

```
lattice> 2 + 2
=> 4
lattice> "hello" + " " + "world"
=> "hello world"
lattice> [1, 2, 3].map(|x| x * 10)
=> [10, 20, 30]
```

The `=>` prefix is how the REPL shows you the result of an expression. Statements that produce no meaningful value—like variable assignments—are silently suppressed:

```
lattice> let name = "Lattice"
lattice> print(name)
Lattice
lattice>
```

The REPL maintains **persistent state**: variables, functions, structs, and enums you define in one line are available on the next. This makes it a wonderful tool for exploratory programming.

```
lattice> fn square(x: Int) -> Int { x * x }
lattice> square(7)
=> 49
lattice> let values = [1, 2, 3, 4, 5]
lattice> values.map(|x| square(x))
=> [1, 4, 9, 16, 25]
```

We will dive much deeper into the REPL in Chapter 2.

1.4.3 Passing Arguments to Scripts

Arguments after the filename are passed to your program and can be retrieved with the `args()` built-in:

Listing 1.7: echo.lat — reading command-line arguments

```
let arguments = args()
print("You passed ${to_string(arguments.len())} arguments:")
for arg in arguments {
    print("  ${arg}")
}
```

```
./clat echo.lat hello world 42
```

Output:

```
You passed 4 arguments:
  echo.lat
  hello
  world
  42
```

The first element is the script's own filename, following the convention of most languages.

1.5 A Taste of What Makes Lattice Different

Before we close this chapter, let's take a quick tour of some features that set Lattice apart. Don't worry about understanding every detail yet—each of these gets a full chapter later. The goal here is to whet your appetite.

1.5.1 The Phase System in Action

Here is the phase system at work in a realistic scenario. Imagine building a server configuration that should be mutable during setup, then locked down before the server starts:

Listing 1.8: Building and freezing a configuration

```
// Build the config in a fluid state
flux config = Map::new()
config["host"] = "0.0.0.0"
config["port"] = 8080
config["max_connections"] = 256

// Lock it down
freeze(config)

// Now config is crystal --- safe to share, impossible to change
print(config["host"])           // 0.0.0.0
print(phase_of(config))        // crystal

// config["port"] = 9090      -- this would error!
```

The `forge` block makes this pattern even more elegant:

Listing 1.9: Forge: build fluid, end crystal

```
fix config = forge {
    flux temp = Map::new()
    temp.set("host", "0.0.0.0")
    temp.set("port", "8080")
    temp.set("debug", "false")
    freeze(temp)
}
// config is now crystal, built in one clean block
print(config.get("host"))  // 0.0.0.0
```

1.5.2 Pattern Matching

Lattice's `match` expression supports literal patterns, ranges, guards, destructuring, and the compiler warns you if your patterns are not exhaustive:

Listing 1.10: Pattern matching

```
fn classify_temperature(temp: Int) -> String {
    match temp {
        x if x < 0 => "freezing",
        0..15 => "cold",
        15..25 => "comfortable",
        25..35 => "warm",
        _ => "hot"
    }
}

print(classify_temperature(22))   // comfortable
print(classify_temperature(-5))   // freezing
print(classify_temperature(38))   // hot
```

1.5.3 Structured Concurrency

Lattice uses `scope` and `spawn` for structured concurrency. Every spawned task must complete before its enclosing scope exits, and values sent across threads must be *frozen*—the phase system ensures data-race safety at the language level:

Listing 1.11: Structured concurrency with channels

```
let results = Channel::new()

scope {
    spawn {
        results.send(freeze("task A complete"))
    }
    spawn {
        results.send(freeze("task B complete"))
    }
}

print(results.recv())   // task A complete (or B, depending on scheduling)
print(results.recv())   // the other one
```

Notice: only `freeze()`'d values can be sent on a channel. This is the phase system doing its job—guaranteeing that shared data is immutable.

1.5.4 Closures and Functional Style

Closures are first-class citizens in Lattice:

Listing 1.12: Closures and higher-order functions

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

let evens = numbers.filter(|n| n % 2 == 0)
let doubled = evens.map(|n| n * 2)
let total = doubled.reduce(|acc, n| acc + n, 0)

print(total)  // 60
```

And you can chain these together more fluidly with the pipe operator:

Listing 1.13: Chaining transforms with iterators

```
let total = iter([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
    .filter(|n| n % 2 == 0)
    .map(|n| n * 2)
    .reduce(0, |acc, n| acc + n)

print(total)  // 60
```

1.5.5 String Interpolation Everywhere

Double-quoted strings support `${expr}` interpolation. Single-quoted strings are raw—no interpolation, no escape sequences. Triple-quoted strings preserve whitespace and support interpolation:

Listing 1.14: String varieties

```
let language = "Lattice"
let version = version()

// Double-quoted: interpolation
print("Running ${language} ${version}")

// Single-quoted: raw, no interpolation
print('The syntax is ${expr}')  // literal ${expr}

// Triple-quoted: multi-line
let banner = """
    ============================
      Welcome to ${language}
      Version: ${version}
    ============================
    """
print(banner)
```

1.5.6 One More Thing: Strict Mode

At the top of any file, you can write `#mode strict` to opt into stricter phase enforcement. In strict mode, the phase checker runs before execution and catches potential issues early:

Listing 1.15: Strict mode

```
#mode strict

flux data = [1, 2, 3]
fix frozen = freeze(data)
// In strict mode, `data` is consumed by freeze().
// Accessing `data` after this line is a compile-time error.
```

Strict mode is entirely opt-in. The default "casual" mode is more forgiving, making it friendlier for scripting and exploration. We will compare the two modes in Chapter 4.

1.6 Exercises

1. **Hello, You.** Write a program that asks for the user's name using `input("What is your name? ")` and prints a personalized greeting using string interpolation.
2. **Temperature Table.** Write a program that prints a conversion table from Fahrenheit to Celsius for every 10 degrees from 0 to 212. Use a `for` loop and the range operator. Hint: `for f in range(0, 220, 10) { ... }`
3. **Explore the REPL.** Launch the REPL (`./clat`) and experiment. Define a function, call it with different arguments, and try to break it by passing the wrong types. What error messages do you get?
4. **Freeze Tag.** In the REPL, create an array with `flux items = [10, 20, 30]`. Push a new value onto it. Then freeze it with `freeze(items)` and try to push again. Read the error message—it will become familiar.
5. **Read an Example.** Read the source of `examples/phase_demo.lat` and run it. Before running, predict what each `print` statement will output. Were you right?

What's Next

You have installed Lattice, written your first program, and seen glimpses of the phase system, pattern matching, and concurrency. In the next chapter, we will spend quality time with the REPL—Lattice's interactive laboratory. The REPL is where you will do most of your exploring, testing ideas, and building intuition for how the language works. Let's set up your lab.

Chapter 2

The REPL Is Your Laboratory

A chemistry lab doesn't start with a production facility. It starts with a bench, some glassware, and the freedom to mix things and see what happens. The Lattice REPL is that bench.

The REPL—**R**ead, **E**valuate, **P**rint, **L**oop—is where you'll spend much of your time when learning Lattice. It's where you test a hypothesis about how a function works, where you prototype a data structure before committing it to a file, and where you debug a tricky expression by poking at it from different angles. In this chapter, we'll learn how to use the REPL effectively and discover the features that make it more than a toy calculator.

2.1 Launching the REPL

Start the REPL by running `clat` with no arguments:

```
./clat
```

You'll see the welcome banner:

```
Lattice v0.3.28 -- crystallization-based programming language
Copyright (c) 2026 Alex Jokela. BSD 3-Clause License.
Type expressions to evaluate. Ctrl-D to exit.
```

The `lattice>` prompt awaits your input. Type an expression and press Enter:

```
lattice> 6 * 7
=> 42
```

To exit, press `Ctrl-D` (which sends an EOF signal) or close the terminal.

2.1.1 Persistent State Across Lines

The REPL is not a series of disconnected calculations. Every variable, function, struct, and enum you define persists for the rest of the session. Think of it as a single, growing program that you write one line at a time:

```
lattice> let city = "Portland"
lattice> let state = "Oregon"
lattice> print("${city}, ${state}")
Portland, Oregon
lattice> let population = 652_503
lattice> population
=> 652503
```

The variable `city` defined on line one is still available on line three. The variable `population` defined on line four is available for the rest of the session.

This is possible because the REPL maintains a single, long-lived virtual machine instance. Under the hood (in `src/main.c`), each line you type is compiled to bytecode and executed on the same `StackVM`—globals, functions, and type definitions are preserved between compilations.

REPL Persistence

The REPL uses a *persistent VM*: a single `StackVM` instance that survives across lines. Variables stored as globals, function definitions, struct declarations, and enum types all remain accessible until you exit. This is what allows you to build programs interactively, one piece at a time.

2.1.2 Multi-line Input

What happens when you type an incomplete expression—like opening a brace without closing it?

```
lattice> fn add(a: Int, b: Int) -> Int {
    ...>     a + b
    ...> }
lattice> add(3, 4)
=> 7
```

The REPL detects that the input is incomplete by checking whether brackets, braces, and parentheses are balanced. When they are not, the prompt changes to `...>` and waits for more input. The accumulated text is only compiled and executed once the delimiters are balanced.

This works for any kind of nesting: function bodies, `if`/`else` blocks, `match` expressions, struct definitions, and more.

```
lattice> let message = if true {
    ...>     "yes"
    ...> } else {
    ...>     "no"
    ...> }
lattice> message
=> "yes"
```

Escaping a Multi-line Mistake

If you start typing a multi-line expression and realize you've made a mistake, press `Ctrl-C` to cancel the current input and return to a fresh `lattice>` prompt. Alternatively, close all open delimiters to force evaluation (which will likely produce an error, but clears the buffer).

2.2 Tab Completion and Discoverability

The Lattice REPL supports tab completion powered by libedit (or readline, depending on your system). Press `Tab` and the REPL will suggest completions based on what you've typed so far.

2.2.1 Keyword and Built-in Completion

Start typing a keyword or built-in function name and press `Tab`:

```
lattice> fr<Tab>
freeze
```

If there are multiple matches, pressing Tab twice shows all options:

```
lattice> to<Tab><Tab>
to_string  to_int  to_float  toml_parse  toml_stringify
```

The completion engine knows about all of Lattice's keywords (`fn`, `let`, `flux`, `fix`, `match`, `struct`, `enum`, `trait`, `impl`, and more) and over 120 built-in functions. This makes the REPL a discovery tool: if you vaguely remember that there's a function related to files, type `file` and press Tab:

```
lattice> file<Tab><Tab>
file_exists  file_size
```

2.2.2 Method Completion After a Dot

The completion engine is context-aware. When the cursor follows a dot (`.`), it switches to *method mode* and suggests method names instead of global keywords:

```
lattice> "hello".to<Tab><Tab>
to_upper  to_lower
```

```
lattice> [1, 2, 3].fi<Tab><Tab>
filter  find  first  flat  flat_map  fill
```

The completion source for methods includes array, string, map, set, channel, buffer, and enum methods—all drawn from the same list defined in `src/completion.c`.

2.2.3 Constructor Completion After ::

Type a type name followed by `::` and press Tab to see available constructors:

```
lattice> Map::<Tab><Tab>
new
lattice> Set::<Tab><Tab>
new  from
lattice> Channel::<Tab><Tab>
new
```

This triple context—global, method, and constructor—means the REPL's tab completion adapts to where you are in an expression.

How Completion Works Internally

The tab completion system is implemented in `src/completion.c`. It registers a custom `rl_attempted_completion_function` with the readline library. When you press `Tab`, this function inspects the character immediately before the cursor:

- If it's a dot (`.`), completion mode switches to `COMPLETE_METHOD`.
- If it's `::`, completion mode switches to `COMPLETE_SCOPE`.
- Otherwise, it uses `COMPLETE_GLOBAL`, which walks through keywords, built-in functions, and constructors.

Filename completion is explicitly disabled—the REPL suggests Lattice symbols, not files.

2.3 Defining Functions, Structs, and Enums Interactively

The REPL is not limited to one-liners. You can define complex types and build up a small program incrementally.

2.3.1 Functions

Define a function and call it immediately:

```
lattice> fn celsius_to_fahrenheit(c: Float) -> Float {
    ...>     c * 9.0 / 5.0 + 32.0
    ...> }
lattice> celsius_to_fahrenheit(100.0)
=> 212.0
lattice> celsius_to_fahrenheit(0.0)
=> 32.0
```

The function persists. You can call it again later, pass it to higher-order functions, or use it inside other definitions:

```
lattice> let temps_c = [0.0, 20.0, 37.0, 100.0]
lattice> temps_c.map(|c| celsius_to_fahrenheit(c))
=> [32.0, 68.0, 98.6, 212.0]
```

2.3.2 Structs

Define a struct, create instances, and call their methods:

```
lattice> struct Point {
    ...>     x: Float,
    ...>     y: Float
    ...> }
lattice> let origin = Point { x: 0.0, y: 0.0 }
lattice> let target = Point { x: 3.0, y: 4.0 }
lattice> target.x
=> 3.0
```

Add a trait and implement it:

```
lattice> trait Measurable {
    ...>     fn distance(self: Any) -> Float
    ...> }
lattice> impl Measurable for Point {
    ...>     fn distance(self: Any) -> Float {
    ...>         sqrt(self.x * self.x + self.y * self.y)
    ...>     }
    ...> }
lattice> target.distance()
=> 5.0
```

2.3.3 Enums

Enums work the same way:

```
lattice> enum Direction {
    ...>     North,
    ...>     South,
    ...>     East,
    ...>     West
    ...> }
lattice> let heading = Direction::North
lattice> heading.variant_name()
=> "North"
lattice> heading.is_variant("North")
=> true
```

```
lattice> enum Shape {
    ...>     Circle(Float),
    ...>     Rectangle(Float, Float)
    ...> }
lattice> let s = Shape::Rectangle(4.0, 5.0)
lattice> s.payload()
=> [4.0, 5.0]
```

The ability to define, test, and iterate on types interactively is one of the REPL's great strengths. You can experiment with a struct's field layout or an enum's variants before committing them to a source file.

2.4 The => Prefix: How the REPL Talks Back

You've already seen the `=>` prefix in front of REPL results. Let's understand exactly when it appears and what it means.

2.4.1 The Display Rule

The REPL follows a clear rule: **if the last expression in your input produces a value that is not `unit` and not `nil`, it is displayed with the `=>` prefix.**

```
lattice> 42
=> 42
lattice> "hello"
=> "hello"
lattice> [1, 2, 3]
=> [1, 2, 3]
lattice> true
=> true
```

Statements that produce `unit` are silently suppressed:

```
lattice> let x = 5
lattice> flux counter = 0
lattice> counter += 1
lattice>
```

No `=>` output. Variable declarations and assignments return `unit`, so there is nothing to display.

The `print()` function writes directly to standard output—it does *not* produce a REPL result:

```
lattice> print("hello")
hello
lattice>
```

Notice: `hello` appears without the `=>` prefix. That's `print()` writing to stdout. The `print()` function itself returns `unit`, so the REPL suppresses its return value.

2.4.2 The `repr` Format

The REPL uses the `repr` format for display, not `to_string`. The difference matters for strings:

```
lattice> "hello world"
=> "hello world"
```

The quotes are part of the display—they tell you this value is a string. If you `print()` the same value, the quotes are absent:

```
lattice> print("hello world")
hello world
```

This distinction between `repr` (for programmers, with type information) and `print` (for users, raw output) is consistent throughout Lattice. Structs that define a custom `repr` closure field will use it for REPL display.

2.4.3 Distinguishing `print` from =>

Here is a common source of confusion for newcomers. What does this produce?

```
lattice> print(2 + 2)
4
lattice>
```

The `4` is from `print()`. The REPL has no `=>` line because `print()` returns `unit`. Compare with:

```
lattice> 2 + 2
=> 4
```

Here the `4` is the REPL displaying the expression result. Understanding this distinction helps you debug REPL sessions: if you see `=>`, it's the value of the expression. If you don't, it's side-effect output (like `print()`).

Unit and Nil

`unit` is the return type of statements and functions with no meaningful return value—like variable assignments or `print()`. `nil` is an explicit null value that means "no value here." Both are suppressed in REPL output, but they are different types: `typeof(nil)` returns `"Nil"` and `typeof(unit)` returns `"Unit"`. We will cover both in Chapter 3.

2.5 Tips and Tricks for Exploratory Programming

The REPL is more than an input box. Here are techniques that will make your interactive sessions more productive.

2.5.1 Use `typeof()` to Inspect Values

When you're not sure what type a value has, ask:

```
lattice> typeof(42)
=> "Int"
lattice> typeof(3.14)
=> "Float"
lattice> typeof("hello")
=> "String"
lattice> typeof([1, 2, 3])
=> "Array"
lattice> typeof(Map::new())
=> "Map"
lattice> typeof(nil)
=> "Nil"
lattice> typeof(true)
=> "Bool"
```

This is especially useful when a function returns an unexpected result and you want to know what you're working with.

2.5.2 Use `phase_of()` to Inspect Phase

Check whether a value is fluid or crystal:

```
lattice> flux items = [1, 2, 3]
lattice> phase_of(items)
=> "fluid"
lattice> freeze(items)
lattice> phase_of(items)
=> "crystal"
```

2.5.3 Build Data Structures Incrementally

The REPL is perfect for building up complex data one step at a time:

```
lattice> flux inventory = Map::new()
lattice> inventory["apples"] = 12
lattice> inventory["bananas"] = 6
lattice> inventory["oranges"] = 8
lattice> inventory
=> {"apples": 12, "bananas": 6, "oranges": 8}
lattice> inventory.keys()
=> ["apples", "bananas", "oranges"]
lattice> inventory.values().sum()
=> 26
```

You can see the state of your data at each step, catch mistakes early, and iterate quickly.

2.5.4 Test Functions with Edge Cases

Once you've defined a function, throw edge cases at it:

```
lattice> fn safe_divide(a: Float, b: Float) -> Float {
    ...>     if b == 0.0 { return 0.0 }
    ...>     a / b
    ...> }
lattice> safe_divide(10.0, 3.0)
=> 3.3333333333333335
lattice> safe_divide(10.0, 0.0)
=> 0.0
lattice> safe_divide(-7.5, 2.5)
=> -3.0
```

The REPL gives you a tight feedback loop for testing behavior before writing formal tests.

2.5.5 Use History to Save Typing

The REPL supports command history via the readline library. Press the **Up** and **Down** arrow keys to navigate through previous inputs. This is handled by libedit/readline and works the same as in bash or Python's interactive interpreter.

Non-empty lines are automatically added to the history buffer, so you can recall and modify earlier definitions without retyping them.

2.5.6 Explore the Standard Library

The REPL is the fastest way to explore Lattice's built-in functions. Wondering what `version()` returns?

```
lattice> version()
=> "0.3.28"
```

Curious about the current platform?

```
lattice> platform()
=> "macos"
```

Want to see what time it is?

```
lattice> time()
=> 1740000000000
lattice> time_format(time(), "%Y-%m-%d %H:%M:%S")
=> "2025-02-19 16:00:00"
```

Need to know your working directory?

```
lattice> cwd()
=> "/Users/alex/projects/lattice"
```

Every built-in function is available in the REPL, and tab completion helps you find them.

2.5.7 Error Recovery

When you make a mistake, the REPL prints an error and keeps going. Your session is not lost:

```
lattice> let x = 42
lattice> x.push(3)
error: push() requires an array, got Int
lattice> x
=> 42
```

The error did not corrupt the REPL state. `x` is still `42`. You can fix your mistake and try again.

This resilience comes from the REPL's error handling code in `src/main.c`: when execution fails, the VM resets its stack and handler state, but globals and definitions are preserved.

2.5.8 The Debugging REPL

For an even more powerful interactive experience, you can embed `breakpoint()` calls in your source files. When the interpreter hits a breakpoint, it drops into an interactive REPL where you can inspect local variables, evaluate expressions in the current scope, and examine the call stack.

Listing 2.1: Using breakpoint() for in-context debugging

```
fn process_order(items: [String], total: Float) {
    let discount = if total > 100.0 { 0.1 } else { 0.0 }
    let final_price = total * (1.0 - discount)
    breakpoint()  // drops into REPL here
    print("Final price: ${to_string(final_price)}")
}

process_order(["widget", "gadget"], 150.0)
```

When the breakpoint is hit, you'll have access to `items`, `total`, `discount`, and `final_price` right there in the REPL. We'll cover the full debugger in Chapter 32 (*Debugging*).

2.5.9 Alternative REPL Backends

By default, the REPL runs on the stack-based bytecode VM. You can switch to the other backends:

```
./clat --tree-walk    # tree-walking interpreter
./clat --regvm        # register-based VM
```

The tree-walking REPL can be useful if you encounter a bytecode compiler bug and need a fallback. The register VM REPL is more experimental. In normal use, the default bytecode VM REPL is the right choice.

The Self-Hosted REPL

Lattice also includes a self-hosted REPL written in Lattice itself (`repl.lat`), built on the `is_complete`, `lat_eval`, and `input` built-ins. You can run it with `./clat repl.lat`. It lacks tab completion and readline integration, but it's an interesting demonstration of the language's metaprogramming capabilities. The C-based REPL described in this chapter is what you should use day-to-day.

2.5.10 A Complete Exploratory Session

Let's tie everything together with a realistic REPL session. Imagine you are exploring Lattice's array methods for the first time:

```
lattice> let scores = [85, 92, 78, 95, 88, 72, 91]
lattice> scores.len()
=> 7
lattice> scores.sort()
=> [72, 78, 85, 88, 91, 92, 95]
lattice> scores.min()
=> 72
lattice> scores.max()
=> 95
lattice> scores.sum()
=> 601
lattice> scores.filter(|s| s >= 90)
=> [92, 95, 91]
lattice> scores.filter(|s| s >= 90).len()
=> 3
lattice> let avg = to_float(scores.sum()) / to_float(scores.len())
lattice> avg
=> 85.85714285714286
lattice> scores.map(|s| if s >= 90 { "A" } else { "B" })
=> ["B", "A", "B", "A", "B", "B", "A"]
lattice> scores.enumerate()
=> [[0, 85], [1, 92], [2, 78], [3, 95], [4, 88], [5, 72], [6, 91]]
```

In under a minute, you've explored sorting, filtering, mapping, aggregation, and enumeration—all without writing a file. That is the power of an interactive session.

2.6 Exercises

1. **REPL Warm-Up.** Launch the REPL and experiment with arithmetic: try `2 ** 10` (does it work? what does Lattice use for exponentiation?), integer vs. float division (`7 / 2` vs. `7.0 / 2.0`), and the modulo operator.

2. **Define and Iterate.** In the REPL, define a function `fn is_palindrome(s: String) -> Bool` that checks whether a string reads the same forwards and backwards. Hint: `s.reverse()`. Test it with `"racecar"`, `"hello"`, and `"madam"`.

3. **Struct Exploration.** Define a `struct Color` with fields `r: Int`, `g: Int`, `b: Int` in the REPL. Create several instances. Then define a function `fn to_hex(c: Any) -> String` that converts a Color to a hex string like `"#FF8800"`. Hint: use `format()` or string concatenation.

4. **Tab Completion Safari.** In the REPL, type `str` and press `Tab`. What completions appear? Type `"hello".` and press `Tab` twice. How many string methods are available? Try `Map::` and `Tab`.

5. **The Mystery Function.** Without looking it up, use the REPL to figure out what `ord("A")` returns. Then figure out what `chr(72)` returns. Can you write a one-liner that converts a string to an array of character codes?

What's Next

Now that you know how to use the REPL as your personal laboratory, it's time to learn what you can put into it. In the next chapter, we will explore Lattice's value types—integers, floats, booleans, strings, nil, and unit—and the operators that work on them. We'll see how Lattice thinks about data at the most fundamental level: the shape of things.

Chapter 3

Values, Types, and the Shape of Things

Before you can freeze something, you need something *to* freeze. Before you can match a pattern, you need data shaped in a way the pattern can describe. Everything in Lattice starts with values—the fundamental atoms of computation—and their types.

In this chapter, we will meet every core value type in Lattice, learn the operators that transform them, and discover the surprisingly rich world of string literals. By the end, you will have a complete mental model of the raw materials Lattice gives you to work with.

3.1 Integers

Integers in Lattice are 64-bit signed values, stored internally as C's `int64_t` (you can verify this in `include/value.h`, where `VAL_INT` maps to the `int_val` field of the value union). This gives you a range from -2^{63} to $2^{63} - 1$—large enough for most practical purposes.

Listing 3.1: Integer basics

```
let population = 8_100_000_000
let negative = -42
let zero = 0

print(typeof(population))  // Int
print(typeof(zero))        // Int
```

Integer Range

Lattice integers are 64-bit signed: they range from $-9{,}223{,}372{,}036{,}854{,}775{,}808$ to $9{,}223{,}372{,}036{,}854{,}775{,}807$. There is no automatic promotion to big integers—if you overflow, you'll get unexpected results. For most applications, 64 bits is more than sufficient.

3.2 Floats

Floating-point numbers use 64-bit double precision (C's `double`). Any number literal containing a decimal point is a float:

Listing 3.2: Float basics

```
let pi = 3.14159
let temperature = -40.0
let ratio = 0.618

print(typeof(pi))           // Float
print(typeof(temperature))  // Float
```

Integer Division vs. Float Division

Division between two integers produces an integer (truncated toward zero). If you want a decimal result, make sure at least one operand is a float:

```
print(7 / 2)       // 3 (integer division)
print(7.0 / 2.0)   // 3.5 (float division)
print(7 / 2.0)     // 3.5 (mixed: promotes to float)
```

Lattice provides two functions for inspecting special float values:

Listing 3.3: Special float values

```
print(is_nan(0.0 / 0.0))   // true
print(is_inf(1.0 / 0.0))   // true
```

You can convert between integers and floats explicitly:

Listing 3.4: Numeric conversions

```
let x = to_float(42)     // 42.0
let y = to_int(3.14)     // 3 (truncates toward zero)
let z = round(3.7)       // 4
let w = floor(3.9)       // 3
let v = ceil(3.1)        // 4
```

3.3 Booleans

Booleans are `true` or `false`. No surprises here:

Listing 3.5: Boolean basics

```
let is_valid = true
let has_errors = false

print(typeof(is_valid))  // Bool
print(!is_valid)         // false
print(is_valid && has_errors)  // false
print(is_valid || has_errors)  // true
```

3.3.1 Truthiness

In contexts where a boolean is expected (like `if` conditions), Lattice applies *truthiness* rules:

- `false` is falsy.
- `nil` is falsy.
- `0` (integer zero) is falsy.
- `""` (empty string) is falsy.
- Everything else is truthy—including empty arrays and empty maps.

Listing 3.6: Truthiness in action

```
if 42 { print("truthy") }         // truthy
if "" { print("truthy") } else { print("falsy") }  // falsy
if nil { print("truthy") } else { print("falsy") }  // falsy
if [] { print("truthy") }         // truthy (empty array is truthy!)
```

Empty Arrays Are Truthy

Unlike Python, empty arrays and empty maps are *truthy* in Lattice. If you want to check whether a collection is empty, use `.len() == 0` or `.is_empty()` (for strings).

3.4 Strings

Strings in Lattice are immutable, UTF-8 encoded sequences of characters. They are the most feature-rich of the primitive types, with three different literal syntaxes and over 20 built-in methods.

3.4.1 Double-Quoted Strings

The standard string literal uses double quotes and supports both escape sequences and interpolation:

Listing 3.7: Double-quoted strings

```
let greeting = "Hello, World!"
let with_newline = "line one\nline two"
let with_tab = "column A\tcolumn B"
```

The full set of escape sequences is:

3.4.2 String Interpolation

Inside double-quoted strings, `${expr}` embeds the result of any expression:

Escape	Character
`\\n`	Newline
`\\t`	Tab
`\\r`	Carriage return
`\\0`	Null byte
`\\\\`	Literal backslash
`\\"`	Literal double quote
`\\'`	Literal single quote
`\\$`	Literal dollar sign (prevents interpolation)
`\\xHH`	Hex byte (e.g., `\\x41` for 'A')

Table 3.1: String escape sequences

Listing 3.8: String interpolation

```
let language = "Lattice"
let major = 0
let minor = 3

print("Welcome to ${language}")
// Welcome to Lattice

print("Version ${major}.${minor}")
// Version 0.3

print("2 + 2 = ${2 + 2}")
// 2 + 2 = 4

let name = "world"
print("${"hello".to_upper()}, ${name}!")
// HELLO, world!
```

The expression inside the braces can be arbitrarily complex—function calls, method chains, arithmetic, even nested strings. The lexer handles this by tracking brace depth, recursively lexing the expression tokens embedded within the string (you can see this machinery in `src/lexer.c`, where interpolated strings are split into `TOK_INTERP_START`, expression tokens, and `TOK_INTERP_END` segments).

To include a literal `${` in a string, escape the dollar sign:

Listing 3.9: Escaping interpolation

```
print("Use \${expr} for interpolation")
// Use ${expr} for interpolation
```

3.4.3 Single-Quoted Strings

Single-quoted strings have **no interpolation**. The `${...}` syntax is treated as literal characters:

Listing 3.10: Single-quoted strings: no interpolation

```
let pattern = 'Hello, ${name}'
print(pattern)
// Hello, ${name}
```

Single-quoted strings still support escape sequences (`\\n`, `\\t`, etc.), so they are not fully "raw" strings. Their primary use case is when you have strings that contain dollar signs and braces and you don't want to worry about accidental interpolation:

Listing 3.11: Single-quoted strings for templates

```
let template = 'Dear ${recipient}, your order #${order_id} is ready.'
print(template)
// Dear ${recipient}, your order #${order_id} is ready.
```

When to Use Single vs. Double Quotes

Use double quotes when you want interpolation: `"Hello, ${name}"`. Use single quotes when you're writing templates, regex patterns, or any string where `$\{...\}` should be literal text.

3.4.4 Triple-Quoted Strings

Triple-quoted strings (`""" ... """`) span multiple lines and support automatic dedenting:

Listing 3.12: Triple-quoted strings

```
let html = """
    <div>
        <h1>Welcome</h1>
        <p>Hello, world!</p>
    </div>
    """
print(html)
```

Output (note how the leading whitespace has been stripped):

```
<div>
    <h1>Welcome</h1>
    <p>Hello, world!</p>
</div>
```

The dedenting algorithm uses the indentation of the closing `"""` as the baseline and strips that much leading whitespace from every line. This lets you indent your multi-line strings to match the surrounding code without polluting the string content.

Triple-quoted strings support interpolation:

Listing 3.13: Triple-quoted strings with interpolation

```
let name = "Lattice"
let version = "0.3.28"
let banner = """
    ============================
     ${name} v${version}
     A crystallization language
    ============================
    """
print(banner)
```

3.4.5 Common String Methods

Strings come with a rich set of methods. Here are some of the most commonly used ones:

Listing 3.14: String methods

```
let message = "  Hello, Lattice!  "

print(message.trim())            // "Hello, Lattice!"
print(message.len())             // 20
print(message.contains("Lat"))   // true
print(message.to_upper())        // "  HELLO, LATTICE!  "
print(message.replace("Lattice", "World"))  // "  Hello, World!  "
print(message.split(","))        // ["  Hello", " Lattice!  "]

let path = "/home/user/documents/report.txt"
print(path.starts_with("/home"))  // true
print(path.ends_with(".txt"))     // true
print(path.index_of("user"))     // 6
```

For a complete reference of string methods, see Chapter 8 (*Strings in Depth*).

3.5 Nil and Unit

Lattice has two "empty" types, and understanding the difference matters.

3.5.1 Nil

`nil` represents "no value here"—the explicit absence of a meaningful value. It's the value returned by `Map.get()` for a missing key, and it's what you use when you want to intentionally store "nothing":

Listing 3.15: Nil basics

```
let nothing = nil
print(typeof(nothing))  // Nil

flux m = Map::new()
m["key"] = "value"
print(m.get("key"))      // value
print(m.get("missing"))  // nil
```

`nil` is *falsy*—it evaluates to false in boolean contexts. This makes it natural to use with the nil coalescing operator:

Listing 3.16: Nil coalescing with ??

```
let port = nil ?? 8080
print(port)  // 8080

let name = "Alice" ?? "Anonymous"
print(name)  // Alice (left side is not nil, so it wins)
```

3.5.2 Unit

`unit` is the type of "nothing happened"—the return type of statements, `print()` calls, and variable assignments. You rarely encounter it explicitly, but it's always there behind the scenes:

Listing 3.17: Unit in action

```
let result = print("hello")
print(typeof(result))  // Unit
```

In the REPL, both `nil` and `unit` are suppressed—neither produces the `=>` output line.

Nil vs. Unit

`nil` means "this value is intentionally empty." It can be stored in variables, returned from functions, and compared with ==.
`unit` means "this expression produces no value." It's the implicit return of statements and void-like functions. You typically don't store it or test for it.
Think of `nil` as an empty box and `unit` as the absence of a box.

3.6 Type Checking with `typeof()`

The `typeof()` built-in function returns a string naming the type of any value:

Listing 3.18: typeof() for every core type

```
print(typeof(42))             // Int
print(typeof(3.14))           // Float
print(typeof(true))           // Bool
print(typeof("hello"))        // String
print(typeof(nil))            // Nil
print(typeof([1, 2, 3]))     // Array
print(typeof(Map::new()))    // Map
print(typeof(Set::new()))    // Set
print(typeof((1, 2)))        // Tuple
print(typeof(0..10))         // Range
print(typeof(|x| x))         // Closure
```

`typeof()` returns the string name, not a type object. You compare it with string equality:

Listing 3.19: Runtime type checking

```
fn describe(value: any) -> String {
    if typeof(value) == "Int" {
        "an integer: ${to_string(value)}"
    } else if typeof(value) == "String" {
        "a string: ${value}"
    } else if typeof(value) == "Array" {
        "an array of length ${to_string(value.len())}"
    } else {
        "something else: ${to_string(value)}"
    }
}

print(describe(42))          // an integer: 42
print(describe("hello"))     // a string: hello
print(describe([1, 2, 3]))  // an array of length 3
```

For struct instances, `typeof()` returns the struct's name:

Listing 3.20: typeof() with structs

```
struct User { name: String, age: Int }
let alice = User { name: "Alice", age: 30 }
print(typeof(alice))  // User
```

Type Annotations vs. typeof()

Function parameter type annotations (`fn foo(x: Int)`) and `typeof()` serve different purposes. Annotations are checked automatically on every call and produce clear error messages. `typeof()` is for when you need to branch on type at runtime—rare in well-typed code, but useful for utility functions, serializers, and debugging.

3.7 Number Literals in Detail

Lattice supports several formats for writing numeric literals, all designed for readability.

3.7.1 Underscore Separators

You can place underscores between digits in both integer and float literals. The underscores are purely visual—they are stripped before the number is parsed:

Listing 3.21: Underscore separators

```
let population = 8_100_000_000
let bytes = 1_048_576
let precise_pi = 3.14_159_265

print(population)   // 8100000000
print(bytes)        // 1048576
print(precise_pi)   // 3.14159265
```

This is especially useful for large numbers where counting zeros is error-prone.

3.7.2 Hexadecimal Literals

Integer literals can be written in hexadecimal with the `0x` prefix:

Listing 3.22: Hexadecimal literals

```
let red = 0xFF0000
let mask = 0xDEAD_BEEF
let byte = 0x7F

print(red)    // 16711680
print(mask)   // 3735928559
print(byte)   // 127
```

Hex digits can be uppercase or lowercase, and underscore separators work with hex too. The lexer (in `src/lexer.c`) detects the `0x` or `0X` prefix, reads hex digits with `isxdigit()`, strips underscores, and converts using `strtoll()` with base 16.

3.7.3 No Scientific Notation

Lattice does not currently support scientific notation (like `1.5e10` or `3e-4`). If you need to express very large or very small numbers, use `pow()`:

Listing 3.23: Expressing large numbers

```
let avogadro = 6.022 * pow(10.0, 23.0)
let planck = 6.626 * pow(10.0, -34.0)
```

3.8 Operators

Lattice provides a comprehensive set of operators that will feel familiar to anyone coming from C, JavaScript, or Python.

3.8.1 Arithmetic Operators

The `+` operator also works for string concatenation:

Listing 3.24: String concatenation with +

```
let full = "Hello" + ", " + "World!"
print(full)  // Hello, World!
```

Operator	Description	Example	Result
`+`	Addition	`3 + 4`	`7`
`-`	Subtraction	`10 - 3`	`7`
`*`	Multiplication	`6 * 7`	`42`
`/`	Division	`15 / 4`	`3`
`%`	Modulo	`17 % 5`	`2`

Table 3.2: Arithmetic operators

3.8.2 Comparison Operators

Operator	Description	Example	Result
`==`	Equal	`3 == 3`	`true`
`!=`	Not equal	`3 != 4`	`true`
`<`	Less than	`3 < 5`	`true`
`>`	Greater than	`5 > 3`	`true`
`<=`	Less or equal	`3 <= 3`	`true`
`>=`	Greater or equal	`4 >= 5`	`false`

Table 3.3: Comparison operators

Comparison operators work on integers, floats, strings (lexicographic order), and booleans. Equality (==) uses structural comparison for arrays, maps, and structs:

Listing 3.25: Structural equality

```
print([1, 2, 3] == [1, 2, 3])   // true
print([1, 2] == [1, 2, 3])      // false
print("abc" < "abd")             // true (lexicographic)
```

3.8.3 Logical Operators

Both `&&` and `||` are short-circuiting: the right operand is only evaluated if needed.

Operator	Description	Example	Result
`&&`	Logical AND	`true && false`	`false`
`||`	Logical OR	`true || false`	`true`
`!`	Logical NOT	`!true`	`false`

Table 3.4: Logical operators

Listing 3.26: Short-circuit evaluation

```
// The second condition is never evaluated
let safe = false && (1 / 0 > 0)
print(safe)  // false

// The second condition is never evaluated
let ok = true || (1 / 0 > 0)
print(ok)    // true
```

3.8.4 Bitwise Operators

Lattice provides the full set of bitwise operations on integers:

Operator	Description	Example	Result
`&`	Bitwise AND	`0xFF & 0x0F`	`15`
`|`	Bitwise OR	`0xF0 | 0x0F`	`255`
`^`	Bitwise XOR	`0xFF ^ 0x0F`	`240`
`~`	Bitwise NOT	`~0`	`-1`
`<<`	Left shift	`1 << 8`	`256`
`>>`	Right shift	`256 >> 4`	`16`

Table 3.5: Bitwise operators

Bitwise operators are essential for systems programming tasks like flag manipulation, color encoding, and protocol parsing:

Listing 3.27: Bitwise operations in practice

```
// Extract RGB components from a 24-bit color
let color = 0xFF8040
let red   = (color >> 16) & 0xFF   // 255
let green = (color >> 8) & 0xFF    // 128
let blue  = color & 0xFF           // 64

print("R=${to_string(red)} G=${to_string(green)} B=${to_string(blue)}")
// R=255 G=128 B=64
```

3.8.5 Compound Assignment

Every arithmetic and bitwise operator has a compound assignment form:

Listing 3.28: Compound assignment operators

```
flux score = 100
score += 10    // score = 110
score -= 5     // score = 105
score *= 2     // score = 210
score /= 3     // score = 70
score %= 30    // score = 10

flux flags = 0xFF
flags &= 0x0F   // flags = 15
flags |= 0xF0   // flags = 255
flags ^= 0x0F   // flags = 240
flags <<= 4     // flags = 3840
flags >>= 8     // flags = 15
```

Compound Assignment Requires `flux`

Compound assignment operators modify the variable in place. This only works with `flux` (mutable) variables. Attempting to use `+=` on a `let` or `fix` binding that refers to a crystal value will produce an error.

3.8.6 The Nil Coalescing Operator

The `??` operator returns the left side if it is not `nil`; otherwise, it returns the right side:

Listing 3.29: Nil coalescing

```
let name = nil ?? "Anonymous"
print(name)  // Anonymous

let title = "Engineer" ?? "Unknown"
print(title)  // Engineer

// Chains: first non-nil wins
let result = nil ?? nil ?? "found it"
print(result)  // found it
```

This is particularly useful when reading from maps, where missing keys return `nil`:

Listing 3.30: Nil coalescing with map lookups

```
flux config = Map::new()
config["host"] = "localhost"

let host = config.get("host") ?? "0.0.0.0"
let port = config.get("port") ?? 8080

print(host)  // localhost
print(port)  // 8080
```

3.8.7 Optional Chaining

The `?.` operator safely navigates through potentially `nil` values. If the receiver is `nil`, the entire chain short-circuits to `nil` instead of producing an error:

Listing 3.31: Optional chaining

```
let user = nil
print(user?.name)           // nil (no error)
print(user?.address?.city)  // nil (chain short-circuits)

// Combine with ?? for defaults
let city = user?.address?.city ?? "Unknown"
print(city)  // Unknown
```

3.8.8 The Range Operator

The `..` operator creates a range—a pair of integers representing a half-open interval (`start` inclusive, `end` exclusive):

Listing 3.32: Range operator

```
for i in 0..5 {
    print(i)  // 0, 1, 2, 3, 4
}

print(typeof(0..10))  // Range
```

Ranges are used with `for` loops and in pattern matching.

3.9 Comments

Lattice supports three styles of comments.

3.9.1 Line Comments

A `//` starts a comment that extends to the end of the line:

Listing 3.33: Line comments

```
let x = 42  // this is a comment
// this entire line is a comment
```

3.9.2 Block Comments

Block comments use `/* ... */` and can span multiple lines. They can be **nested**, which is particularly useful for commenting out code that already contains comments:

Listing 3.34: Block comments (nestable)

```
/* This is a block comment */

/*
 * Multi-line block comment.
 * Often used for function documentation.
 */

/* You can /* nest */ block comments */
```

The nesting support is implemented by tracking a depth counter in the lexer—each `/*` increments it and each `*/` decrements it. The comment only ends when the depth reaches zero.

3.9.3 Doc Comments

Lines starting with `///` are *doc comments*. They are treated as regular comments by the interpreter, but the `clat doc` tool extracts them to generate documentation:

Listing 3.35: Doc comments

```
/// Calculate the area of a circle.
/// Takes a radius and returns the area.
fn circle_area(radius: Float) -> Float {
    3.14159 * radius * radius
}
```

We'll cover the documentation generator in detail in Chapter 30 (*The Formatter and Doc Generator*). For now, get in the habit of documenting your functions with `///`—your future self will thank you.

3.10 Putting It All Together

Let's write a small program that uses several of the types and operators we've covered:

Listing 3.36: A temperature converter using multiple types

```
/// Convert a temperature between Fahrenheit and Celsius.
/// The `from` parameter must be "F" or "C".
fn convert_temp(value: Float, from: String) -> String {
    let result = if from == "F" {
        (value - 32.0) * 5.0 / 9.0
    } else {
        value * 9.0 / 5.0 + 32.0
    }

    let to_unit = if from == "F" { "C" } else { "F" }
    let rounded = round(result * 100.0) / 100.0

    "${to_string(value)} ${from} = ${to_string(rounded)} ${to_unit}"
}

// Build a conversion table
let temps = [0.0, 32.0, 72.0, 100.0, 212.0]
for temp in temps {
    print(convert_temp(temp, "F"))
}
/*
    0.0 F = -17.78 C
    32.0 F = 0.0 C
    72.0 F = 22.22 C
    100.0 F = 37.78 C
    212.0 F = 100.0 C
*/
```

This program demonstrates:

- **Floats** for temperature values
- **Strings** with interpolation for formatted output
- **Comparison operators** (==) for branching
- **Arithmetic operators** for the conversion formula
- **Arrays** to hold a list of temperatures
- **Doc comments** on the function
- **Block comments** for expected output

- The `if`/`else` expression returning a value

3.11 Exercises

1. **Type Explorer.** In the REPL, use `typeof()` on at least 10 different values, including an array of arrays, a map with string keys, a range, a closure, and `nil`. Write down any surprises.
2. **Hex Color Parser.** Write a function `fn parse_color(hex: Int) -> [Int]` that takes a hex color like `0xFF8040` and returns an array `[r, g, b]` using bitwise operators. Test it with several colors.
3. **String Processing.** Write a function that takes a full name like `"ada lovelace"` and returns it in title case: `"Ada Lovelace"`. Use string methods. (Hint: `.title_case()` exists!)
4. **Nil Safety.** Write a function `fn safe_get(m: Map, key: String, default_val: any) -> any` that returns the map's value for the given key, or `default_val` if the key is missing. Use the `??` operator.
5. **Triple-Quote Art.** Use a triple-quoted string with interpolation to create an ASCII art box around a message. The function should take a message string and return the boxed version with proper padding.

What's Next

We've surveyed the raw materials—integers, floats, booleans, strings, nil, and unit—and the tools for working with them. But so far, our values have been rootless: created, used, and forgotten. In the next chapter, we will learn how to *bind* values to names with `let`, `flux`, and `fix`, and we'll get our first real look at the phase system that gives Lattice its crystalline character.

Chapter 4

Variables and the Idea of Phase

In most languages, declaring a variable is a mundane act. You pick a name, assign a value, and move on. In Lattice, declaring a variable is a *decision*—a declaration of intent about how that value should behave over its lifetime. Will it change? Should it be locked down? Or should the language figure that out for you?

Lattice gives you three keywords for variable declaration—`let`, `flux`, and `fix`—and each one makes a different promise to the runtime. Behind these keywords lies the *phase system*, the feature that gives Lattice its name and its crystalline metaphor. This chapter introduces both the practical mechanics of variable binding and the philosophy that makes them meaningful.

4.1 `let`, `flux`, and `fix`

4.1.1 `let` — Inferred Phase

The `let` keyword declares a variable and infers its phase from the value you assign:

Listing 4.1: Declaring variables with let

```
let name = "Lattice"      // a string
let version = 28           // an integer
let pi = 3.14159           // a float
let active = true          // a boolean
let colors = ["red", "green", "blue"]  // an array
```

With `let`, the variable's mutability depends on what you assign. In casual mode (the default), `let` bindings are treated as fluid—you *can* reassign them or mutate their contents:

Listing 4.2: let bindings are fluid in casual mode

```
let score = 100
// In casual mode, this works:
// score = 200

let items = [1, 2, 3]
items.push(4)  // modifying in place works
print(items)   // [1, 2, 3, 4]
```

Think of `let` as the "just give me a variable" keyword—quick, convenient, and unassuming. It's perfect for scripts, quick prototypes, and situations where you don't need to think about mutability.

4.1.2 `flux` — Explicitly Fluid

The `flux` keyword (pronounced like "flux" in metallurgy) declares a variable that is *explicitly fluid*. A fluid value can be modified, reassigned, grown, and reshaped:

Listing 4.3: Declaring mutable variables with flux

```
flux counter = 0
counter += 1
counter += 1
print(counter)  // 2

flux inventory = Map::new()
inventory["apples"] = 12
inventory["bananas"] = 6
print(inventory)  // {"apples": 12, "bananas": 6}

flux temperatures = []
temperatures.push(72.1)
temperatures.push(68.5)
temperatures.push(74.3)
print(temperatures)  // [72.1, 68.5, 74.3]
```

The `flux` prefix is a signal to anyone reading your code: "this variable *will* change." When you see `flux` in a program, you know to watch for mutations.

Fluid Phase

A value in the *fluid phase* is mutable. Arrays can be pushed to, maps can gain new keys, and the variable can be reassigned. In the chemistry metaphor, a fluid value is like molten metal—it has not yet solidified into its final form. You declare fluid variables with `flux` (or the shorthand prefix ~).

Under the hood, the compiler emits an `OP_MARK_FLUID` instruction when it encounters `flux`, which sets the value's phase tag to `VTAG_FLUID` (you can see this in `src/stackcompiler.c`). This tag travels with the value for its entire lifetime.

4.1.3 `fix` — Explicitly Crystal

The `fix` keyword declares a variable that is *crystal*—immutable, locked, permanent. The value must already be frozen (or will be frozen by the declaration):

Listing 4.4: Declaring immutable variables with fix

```
fix pi = freeze(3.14159)
fix greeting = freeze("Hello, World!")
fix primes = freeze([2, 3, 5, 7, 11])

print(pi)        // 3.14159
print(greeting)  // Hello, World!
print(primes)    // [2, 3, 5, 7, 11]
```

Attempting to modify a crystal value produces an error:

Listing 4.5: Crystal values reject modification

```
fix config = freeze([1, 2, 3])
// config.push(4)  -- error: cannot mutate crystal value
// config = [4, 5, 6]  -- error: cannot assign to crystal binding
```

Crystal Phase

A value in the *crystal phase* is immutable. Its contents cannot be modified, its structure cannot change, and it can be safely shared across threads. In the chemistry metaphor, a crystal is solid, ordered, and permanent—like a diamond formed under pressure. You declare crystal variables with `fix` (or the shorthand prefix `*`).

When the compiler encounters `fix`, it emits an `OP_FREEZE` instruction that transitions the value to the crystal phase.

4.1.4 The Shorthand Prefixes: ~ and *

In type annotations and function parameters, you'll sometimes see ~ for fluid and * for crystal:

Listing 4.6: Phase shorthand in function parameters

```
fn mutate(data: ~Map) {
    data.set("modified", true)
}

fn inspect(data: *Map) {
    print(data.get("name"))
}
```

Here, `~Map` means "a Map that must be fluid" and `*Map` means "a Map that must be crystal." The runtime checks these constraints on every call—pass a crystal map to `mutate()` and you'll get a clear error.

These shorthand prefixes are equivalent to writing `flux Map` and `fix Map` in type position. Use whichever reads better to you.

4.1.5 Comparing the Three Keywords

Keyword	Phase	Mutable?	When to Use
`let`	Inferred	Depends on mode	Scripts, quick code, "just works"
`flux`	Fluid	Yes	Values you intend to modify
`fix`	Crystal	No	Constants, shared data, configs

Table 4.1: Variable declaration keywords

A rule of thumb: if you're writing a script or prototyping, `let` is fine. If you're writing production code or library code, prefer `flux` and `fix` for clarity. In strict mode, `let` is actually *disallowed*—you must choose explicitly.

4.2 Why Mutability Is a First-Class Concept

Most languages treat mutability as a binary toggle. In Rust, you have `let` vs. `let mut`. In JavaScript, you have `const` vs. `let`. In Python, everything is mutable and you just hope for the best.

Lattice goes further. Mutability is not just a property of a *variable*—it's a property of the *value itself*. This distinction is subtle but profound.

Listing 4.7: Phase is a property of the value, not the variable

```
flux items = [1, 2, 3]
print(phase_of(items))   // fluid

freeze(items)
print(phase_of(items))   // crystal

// The variable `items` still exists.
// But the *value* it holds has changed phase.
// items.push(4)  -- error: the value is crystal
```

The phase travels with the value. If you pass a frozen array to a function, that function cannot mutate it—even if it assigns it to a new `flux` variable. The crystal phase is embedded in the value, not in the binding.

Listing 4.8: Phase follows the value through function calls

```
fn try_to_modify(data: any) {
    flux local = data
    // local.push(99)  -- error if data is crystal!
    print(phase_of(local))
}

flux numbers = [1, 2, 3]
try_to_modify(numbers)    // prints: fluid

freeze(numbers)
try_to_modify(numbers)    // prints: crystal (push would fail)
```

This design has a powerful consequence: when a value is frozen, it is safe to share it across threads, pass it to unknown functions, or store it in a long-lived data structure—because *nothing* can modify it. The guarantee is enforced by the runtime, not by programmer discipline.

The Value vs. the Binding

The `flux`/`fix` keywords control two things:

1. Whether the **variable binding** can be reassigned (pointing to a different value).
2. The initial **phase of the value** (fluid or crystal).

The second property persists even after the value is passed to other functions or stored in data structures. The first property only applies to the original variable name in its scope.

4.3 Quick Introduction to `freeze()` and `thaw()`

We've already seen `freeze()` in passing. Let's give it proper attention.

4.3.1 `freeze()` — Fluid to Crystal

`freeze()` transitions a value from the fluid phase to the crystal phase. Once frozen, the value is immutable:

Listing 4.9: Freezing a value

```
flux settings = Map::new()
settings["debug"] = false
settings["verbose"] = true
settings["max_retries"] = 3

print(phase_of(settings))  // fluid
freeze(settings)
print(phase_of(settings))  // crystal

// settings["debug"] = true  -- error: cannot mutate crystal value
```

For collections (arrays, maps, sets), `freeze()` is **deep**: it freezes the container and all values inside it recursively.

Listing 4.10: Deep freeze

```
flux data = [[1, 2], [3, 4]]
freeze(data)

// data.push([5, 6])  -- error: outer array is crystal
// data[0].push(3)    -- error: inner array is also crystal
```

4.3.2 thaw() — Crystal to Fluid

thaw() creates a **mutable copy** of a crystal value. The original remains frozen:

Listing 4.11: Thawing a frozen value

```
fix original = freeze([10, 20, 30])
print(phase_of(original))  // crystal

flux copy = thaw(original)
print(phase_of(copy))      // fluid

copy.push(40)
print(copy)      // [10, 20, 30, 40]
print(original)  // [10, 20, 30] (unchanged!)
```

The metaphor is precise: thawing a crystal doesn't destroy the crystal. It melts a copy, leaving the original intact. This is a key property for safe concurrent programming—multiple threads can thaw the same frozen data independently without interfering with each other.

4.3.3 clone() — Independent Copy

While we're talking about copying, let's mention clone(). Unlike thaw(), which always produces a fluid copy from a crystal, clone() creates an independent deep copy of any value, preserving its current phase:

Listing 4.12: Cloning preserves phase

```
flux original = [1, 2, 3]
flux copy = clone(original)

copy.push(4)
print(original)  // [1, 2, 3] (unaffected)
print(copy)      // [1, 2, 3, 4]
```

Function	Input Phase	Output Phase
`freeze(v)`	Fluid	Crystal (same value, now locked)
`thaw(v)`	Crystal	Fluid (new copy)
`clone(v)`	Any	Same as input (new copy)

Table 4.2: Phase transition functions

We'll explore the full depth of phase transitions—including `sublimate`, `crystallize`, `borrow`, `forge`, bonds, reactions, and more—in Part III of this book. For now, `freeze()`, `thaw()`, and `clone()` are the essential trio.

4.4 `#mode casual` vs. `#mode strict`

Lattice offers two modes of phase enforcement, controlled by a directive at the top of a file:

Listing 4.13: Mode directives

```
#mode casual   // the default: relaxed phase rules
#mode strict   // tighter enforcement: no let, consume on freeze
```

If you don't specify a mode, the default is `casual`.

4.4.1 Casual Mode

In casual mode, Lattice is forgiving:

- `let` is allowed and produces an inferred-phase binding.
- `freeze()` transitions a value in place but does *not* consume the variable—you can still read from the binding.

- Phase mismatches at function call boundaries produce runtime errors, not compile-time errors.
- You can mix `let`, `flux`, and `fix` freely.

Listing 4.14: Casual mode: forgiving

```
let data = [1, 2, 3]
data.push(4)         // works: let is fluid in casual mode
freeze(data)
print(data)          // works: data is still accessible (now crystal)
// data.push(5)      -- error: crystal value
```

Casual mode is designed for scripting, exploratory programming, and situations where you want the phase system's safety net without its full strictness.

4.4.2 Strict Mode

Strict mode tightens the rules:

- `let` is **disallowed**. You must use `flux` or `fix` for every declaration, making your intent explicit.
- `freeze()` **consumes the binding**. After freezing a variable, the original name is no longer accessible—you must capture the frozen result in a new binding.
- The phase checker runs **before execution**, catching phase errors as compile-time warnings rather than runtime surprises.
- Assigning to a crystal binding is a compile-time error.

Listing 4.15: Strict mode: explicit and consuming

```
#mode strict

flux data = [1, 2, 3]
data.push(4)

fix frozen = freeze(data)
// `data` is no longer accessible in strict mode.
// You must use `frozen` from now on.
print(frozen)  // [1, 2, 3, 4]
```

The consume-on-freeze behavior prevents a common bug: modifying a value after you've shared a "frozen" reference to it. In casual mode, you could freeze a value, hand the frozen reference to another part of the program, and then accidentally mutate the original. Strict mode makes this impossible by removing the original name from scope.

Listing 4.16: Strict mode catches errors early

```
#mode strict

// let x = 42  -- error: use flux or fix instead of let

flux counter = 0
counter += 1

fix pi = freeze(3.14159)
// pi = 2.71828  -- error: cannot assign to crystal binding

// This error is caught before the program runs.
```

When to Use Strict Mode

Use `#mode strict` when:

- You're writing a library that will be used by others.
- You're writing code that runs in production.
- You're working on concurrent code where phase safety is critical.
- You want the compiler to catch mistakes before runtime.

Use casual mode (the default) when:

- You're scripting or prototyping.
- You're exploring in the REPL.
- You want to move fast without ceremony.

4.4.3 Inside the Phase Checker

When you use `#mode strict`, the parser sets the program's mode to `MODE_STRICT`, which triggers the phase checker (`src/phase_check.c`) before execution begins. The phase checker is a static analysis pass that walks the AST and:

1. Tracks the phase of every variable through a scope stack.
2. Verifies that `fix` bindings are never assigned to.

3. Ensures that arguments passed to phase-constrained parameters (`~Map`, `*Map`) have compatible phases.
4. Rejects `let` bindings—requiring explicit `flux` or `fix`.
5. Inside `spawn` blocks, checks that fluid bindings from the outer scope are not used across the thread boundary.

The phase checker is defined as a `PhaseChecker` struct that maintains a stack of scopes, each mapping variable names to their known phases (`PHASE_FLUID`, `PHASE_CRYSTAL`, or `PHASE_UNSPECIFIED`). As it walks the AST, it pushes and pops scopes at function and block boundaries, just like the runtime would.

The errors it produces are not fatal—they're warnings that appear before execution. But they are informative:

```
phase error: strict mode: use 'flux' or 'fix' instead of 'let' for binding 'x'
phase error: strict mode: cannot assign to crystal binding 'config'
phase error: strict mode: cannot use fluid binding 'data' across thread boundary in
    spawn
```

4.5 First Encounter with the Phase Philosophy

Now that you've seen the mechanics—`flux`, `fix`, `freeze()`, `thaw()`, casual mode, strict mode—let's step back and think about *why* Lattice works this way.

4.5.1 The Lifecycle of Data

Most data in a program goes through a predictable lifecycle:

1. **Construction**: The data is built up incrementally. You add items to a list, set fields in a configuration, accumulate results.
2. **Stabilization**: At some point, the data is "done." The configuration is complete, the list has all its items, the computation is finished.
3. **Consumption**: The stabilized data is read, shared, serialized, sent to another thread, or stored for later use.

In most languages, there is no formal boundary between these stages. You just *stop modifying* the data and hope nobody else modifies it later. This works fine in small programs. In large programs, with shared state and concurrent threads, it becomes a source of bugs.

Lattice makes the boundary explicit:

Listing 4.17: The lifecycle of data in Lattice

```
// 1. Construction (fluid)
flux report = Map::new()
report["title"] = "Q4 Revenue"
report["total"] = 1_250_000
report["approved"] = false

// 2. Stabilization (freeze)
report["approved"] = true
freeze(report)

// 3. Consumption (crystal, safe to share)
print(report.get("title"))     // Q4 Revenue
// send_to_archive(report)     // safe: report is immutable
// spawn { process(report) }   // safe: can be sent across threads
```

The `freeze()` call is the boundary. Before it, the data is under construction. After it, the data is permanent. The runtime enforces this boundary—you can't go back to the construction phase (unless you explicitly `thaw()` a copy).

4.5.2 The Chemistry Metaphor

The terminology in Lattice is deliberately drawn from chemistry and materials science:

- **Fluid** values are like molten metal—shapeable, hot, unstable.
- **Crystal** values are like a diamond or quartz—solid, ordered, permanent.
- `freeze()` is *crystallization*—the transition from fluid to solid.
- `thaw()` is *melting*—creating a fluid copy from a solid.
- `forge` blocks are a *forge*—a controlled environment where you heat, shape, and cool a material in a single operation.

This metaphor extends further in Part III, where we'll meet:

- **Sublimation**: shallow freezing that locks the structure but leaves inner values mutable.
- **Bonds**: links between values that cause one to freeze when another does.
- **Reactions**: callbacks that fire when a value changes phase.

- **Pressure**: restrictions on what can change without fully freezing.
- **Alloys**: structs with per-field phase declarations.
- **Arenas**: memory regions where crystal values are stored.

But even without these advanced features, the core idea is powerful: **make the lifecycle of your data visible in the code**.

4.5.3 Phase and Concurrency

The phase system's biggest payoff comes with concurrency. Lattice's `scope`/`spawn` structured concurrency model requires that any value sent across a channel must be *crystal*. This is enforced at runtime:

Listing 4.18: Phase and concurrency

```
let ch = Channel::new()

flux data = [1, 2, 3]
// ch.send(data)  -- error: cannot send fluid value

freeze(data)
ch.send(data)  // works: crystal values are safe to share

scope {
    spawn {
        let received = ch.recv()
        print(received)  // [1, 2, 3]
    }
}
```

This constraint means that data races—the most common and hardest-to-debug class of concurrency bugs—are impossible in Lattice. If a value can cross a thread boundary, it is guaranteed to be immutable. If it is mutable, it cannot cross. The phase system draws a bright line between "private, mutable, single-threaded" and "shared, immutable, multi-threaded."

We'll explore concurrency in full in Part V. For now, the key insight is: the phase system is not just about preventing accidental mutation. It's a foundation for safe concurrency.

4.5.4 A Preview: The Forge Block

One of the most elegant patterns in Lattice is the `forge` block, which encapsulates the entire lifecycle of a value—construction, stabilization, and output—in a single expression:

Listing 4.19: Forge: build fluid, end crystal

```
fix server_config = forge {
    flux temp = Map::new()
    temp.set("host", "0.0.0.0")
    temp.set("port", "443")
    temp.set("tls", "true")
    temp.set("max_connections", "1024")
    freeze(temp)
}

// server_config is crystal, ready to share
print(server_config.get("host"))  // 0.0.0.0
print(phase_of(server_config))    // crystal
```

Inside the `forge` block, you work with fluid data. The block's result is automatically treated as crystal. The temporary mutable state never escapes. This pattern is so useful that you'll see it throughout Lattice codebases wherever immutable configurations, frozen caches, or thread-safe snapshots are built.

4.6 Practical Patterns

Let's close with some practical patterns that show how `let`, `flux`, `fix`, and `freeze()` work together in real code.

4.6.1 Accumulate Then Freeze

Build a collection incrementally, then freeze it for safe consumption:

Listing 4.20: Accumulate then freeze

```
fn load_user_ids(filename: String) -> [Int] {
    let content = read_file(filename)
    let lines = content.split("\n")
    flux ids = []
    for line in lines {
        let trimmed = line.trim()
        if !trimmed.is_empty() {
            ids.push(parse_int(trimmed))
        }
    }
    freeze(ids)
    return ids
}
```

The function returns a frozen array. Callers can trust that the data will not change out from under them.

4.6.2 Configuration Objects

Build a configuration with sensible defaults, allow overrides, then lock it down:

Listing 4.21: Configuration with defaults

```
fn make_config(overrides: Map) -> Map {
    flux cfg = Map::new()

    // Defaults
    cfg["timeout"] = 30
    cfg["retries"] = 3
    cfg["verbose"] = false

    // Apply overrides
    for entry in overrides.entries() {
        cfg[entry[0]] = entry[1]
    }

    freeze(cfg)
    return cfg
}

flux opts = Map::new()
opts["timeout"] = 60
opts["verbose"] = true

fix config = make_config(opts)
print(config.get("timeout"))   // 60
print(config.get("retries"))   // 3
print(phase_of(config))        // crystal
```

4.6.3 The Fluid Working Copy

When you need to modify frozen data, `thaw()` gives you a working copy:

Listing 4.22: Working with a thawed copy

```
fix original_scores = freeze([85, 92, 78, 95, 88])

// Create a working copy to add a new score
flux updated = thaw(original_scores)
updated.push(91)
updated = updated.sort()
freeze(updated)

print(original_scores)  // [85, 92, 78, 95, 88] (unchanged)
print(updated)          // [78, 85, 88, 91, 92, 95]
```

This immutable-data-with-copies pattern will feel familiar if you've worked with functional programming languages or immutable data structures in Clojure or Elm.

4.7 Exercises

1. **Phase Detective.** In the REPL, create variables with `let`, `flux`, and `fix`. Use `phase_of()` to check the phase of each. Freeze a `flux` variable and check again. Thaw it and check once more. Write down what you observe.

2. **Strict Mode Workout.** Create a file with `#mode strict` at the top. Try using `let` to declare a variable. Try assigning to a `fix` binding. Try freezing a value and then accessing the original name. Record the error messages you see.

3. **Config Builder.** Write a function that builds a database configuration (host, port, database name, connection pool size) using a `forge` block. The function should accept a map of overrides and return a crystal map with defaults merged in.

4. **Freeze Guard.** Write a function `fn ensure_frozen(value: any) -> any` that returns the value unchanged if it's already crystal, or freezes it if it's fluid. Use `phase_of()` to decide.

5. **The Thaw Game.** Start with `fix data = freeze([10, 20, 30])`. Thaw it, add elements, and freeze again. Repeat this three times, building a longer and longer array. Verify that each intermediate frozen version is independent.

What's Next

You now understand how Lattice thinks about variables and mutability. The `let`/`flux`/`fix` keywords give you explicit control over the lifecycle of your data, and the phase system enforces that control

at runtime. We've barely scratched the surface of what the phase system can do—bonds, reactions, sublimation, pressure, and alloys await in Part III.

But first, we need to learn how to *direct* the flow of our programs. In the next chapter, we'll explore Lattice's control flow constructs: `if`/`else`, `while`, `for`, `loop`, and the powerful `match` expression. These are the tools that turn sequences of values into programs that make decisions.

Part II

The Working Programmer

Chapter 5

Control Flow

Every program needs to make decisions. Should we charge the customer or show an error? Has the sensor reading crossed a threshold? Are there more items to process? Control flow is how a program answers these questions and acts accordingly.

What makes Lattice's control flow distinctive is a commitment to *expressions over statements*. In many languages, `if`/`else` is a statement—it does something, but it doesn't produce a value. In Lattice, `if`/`else` is an expression: it evaluates to a result you can bind, pass, or return. This seemingly small design choice ripples through the entire language, making code more concise and composable.

Let's see what that looks like in practice.

5.1 if/else as Expressions

Here is the most basic conditional in Lattice:

Listing 5.1: A basic if/else

```
let temperature = 38

if temperature > 37 {
  print("Fever detected")
} else {
  print("Temperature normal")
}
// Output: Fever detected
```

Nothing surprising so far—this looks like most C-family languages. But watch what happens when we use `if` on the right side of a binding:

Listing 5.2: if/else as an expression

```
let temperature = 38

let status = if temperature > 37 {
  "fever"
} else {
  "normal"
}

print(status) // Output: fever
```

The `if`/`else` block evaluated to `"fever"`, and that value was bound to `status`. There is no need for a ternary operator—`if`/`else` *is* the ternary operator.

Expressions vs. Statements

An *expression* produces a value. A *statement* performs an action. In Lattice, `if`/`else`, `match`, and blocks are all expressions. The last expression in a branch becomes the value of the entire construct.

5.1.1 How the Value Is Determined

The rule is straightforward: the *last expression* in whichever branch executes becomes the value of the whole `if`/`else`. You don't write `return`—the value flows naturally:

Listing 5.3: Multi-line branches still produce values

```
let age = 25

let category = if age < 13 {
  let label = "child"
  label
} else if age < 20 {
  let label = "teenager"
  label
} else {
  let label = "adult"
  label
}

print(category) // Output: adult
```

Each branch can contain multiple statements, but only the final expression contributes the value. Intermediate bindings like `let label` are scoped to their branch and cleaned up automatically.

5.1.2 What Happens Without an else?

If you omit the `else` branch, the expression evaluates to `nil` when the condition is false:

Listing 5.4: Missing else produces nil

```
let temperature = 36

let warning = if temperature > 37 {
  "fever detected"
}

print(warning) // Output: nil
```

nil from Missing else Branches

When you use `if` as an expression without an `else`, the "false" path evaluates to `nil`. This is fine if you expect it, but can be surprising if you forget the `else` branch. If you want a guaranteed non-nil result, always include both branches.

Under the hood, the compiler (in `src/stackcompiler.c`) emits an `OP_JUMP_IF_FALSE` instruction for the condition, compiles both branches, and uses `OP_JUMP` to skip the else branch when the then-path executes. When no `else` is present, the compiler emits an `OP_NIL` instruction as the fallback value. When a branch ends with an expression, the compiler uses `end_scope_preserve_tos` to keep that value on the stack while cleaning up branch-local variables.

5.1.3 Nested and Chained Conditions

You can chain `else if` branches as deeply as you need:

Listing 5.5: Chained else if

```
fn classify_bmi(bmi: Float) -> String {
  if bmi < 18.5 {
    "underweight"
  } else if bmi < 25.0 {
    "normal"
  } else if bmi < 30.0 {
    "overweight"
  } else {
    "obese"
  }
}

print(classify_bmi(22.3)) // Output: normal
```

Because each branch is an expression, the entire chain acts as a single expression that evaluates to one string. No explicit `return` needed—the function's body is the `if`/`else` chain, and its result becomes the return value.

When to Use if/else vs. match

If you're testing a single variable against many values, or destructuring data, `match` (Chapter 9) is usually clearer. Use `if`/`else` for boolean conditions and two-way branches.

5.1.4 Blocks as Expressions

The expression-oriented design extends to plain blocks too. A block enclosed in `{ }` evaluates to its last expression:

Listing 5.6: Block expressions

```
let hypotenuse = {
  let a = 3.0
  let b = 4.0
  (a * a + b * b)
}

print(hypotenuse) // Output: 25.0
```

Block expressions are useful for computing a value that requires intermediate bindings you don't want to leak into the surrounding scope. The variables `a` and `b` exist only within the block. If a block ends with a statement rather than an expression (for instance, if the last line is a `let` binding), the block evaluates to `unit`.

5.2 Loops: while, loop, for

Lattice provides three loop constructs, each suited to a different situation.

5.2.1 while Loops

A `while` loop repeats its body as long as a condition is true:

Listing 5.7: Counting with while

```
flux count = 0

while count < 5 {
  print(count)
  count = count + 1
}
// Output: 0 1 2 3 4
```

Notice the `flux` binding—since we need to mutate `count` on each iteration, it must be declared in the fluid phase. A `let` or `fix` binding cannot be reassigned, so the compiler would reject `count = count + 1`.

The compiler translates a `while` loop into a tight bytecode sequence: compile the condition, emit `OP_JUMP_IF_FALSE` to skip the body, compile the body, then emit `OP_LOOP` to jump back to the condition

check. The `OP_LOOP` instruction uses a backwards jump offset, making the loop machinery compact and efficient.

Listing 5.8: Building a string with while

```
let words = ["the", "quick", "brown", "fox"]
flux result = ""
flux i = 0

while i < words.len() {
  if i > 0 {
    result = result + " "
  }
  result = result + words[i]
  i = i + 1
}

print(result) // Output: the quick brown fox
```

5.2.2 Infinite Loops with loop

When you want a loop that runs until explicitly stopped, use `loop`:

Listing 5.9: An infinite loop with break

```
flux attempts = 0

loop {
  attempts = attempts + 1
  if attempts >= 3 {
    print("Giving up after 3 attempts")
    break
  }
  print("Attempt ${attempts}")
}
// Output:
// Attempt 1
// Attempt 2
// Giving up after 3 attempts
```

A `loop` is semantically equivalent to `while true`, but it communicates intent more clearly: this loop runs forever unless something inside it says otherwise. The compiler generates almost identical bytecode—it omits the condition check entirely, jumping back unconditionally with `OP_LOOP`.

`loop` is especially useful for retry logic, event processing, and REPL-like read-eval-print cycles:

Listing 5.10: A retry pattern with loop

```
fn fetch_with_retry(url: String, max_retries: Int) -> String {
  flux retries = 0
  loop {
    let response = http_get(url)
    if response != nil {
      return response
    }
    retries = retries + 1
    if retries >= max_retries {
      return "failed after ${max_retries} retries"
    }
  }
}
```

5.2.3 for Loops and Iteration

The `for` loop iterates over collections and ranges:

Listing 5.11: Iterating over an array

```
let fruits = ["apple", "banana", "cherry"]

for fruit in fruits {
  print("I like ${fruit}")
}
// Output:
// I like apple
// I like banana
// I like cherry
```

The `for` loop works with any iterable value: arrays, ranges, strings (iterating over characters), maps (iterating over key-value pairs), and sets.

Listing 5.12: Iterating over a range

```
for n in 0..5 {
  print(n)
}
// Output: 0 1 2 3 4
```

Under the hood, the compiler uses two dedicated opcodes for iteration. First, `OP_ITER_INIT` converts the iterable (array, range, etc.) into an internal iterator state that tracks the collection and the current index. Then at the top of each iteration, `OP_ITER_NEXT` pushes the next value onto the stack or jumps past the loop body when the iterator is exhausted. You can see this machinery in `src/stackcompiler.c`—the compiler reserves two anonymous local slots for the iterator state (the collection reference and the index counter) and binds the loop variable as a named local on each iteration.

Listing 5.13: Iterating with an index

```
let languages = ["Lattice", "Rust", "Python", "Go"]

for i in 0..languages.len() {
  print("${i}: ${languages[i]}")
}
// Output:
// 0: Lattice
// 1: Rust
// 2: Python
// 3: Go
```

Listing 5.14: Iterating over a string's characters

```
let greeting = "Hello"

for ch in greeting.chars() {
  print(ch)
}
// Output: H e l l o
```

The Loop Variable Is Fresh Each Iteration

The loop variable (like `fruit` or `n` above) is a new binding on every iteration. You cannot assign to it from outside the loop body, and closures captured inside the loop each see their own copy of that iteration's value.

5.2.4 Nested Loops

Loops nest naturally. Each loop manages its own iterator state independently:

Listing 5.15: Nested loops for a multiplication table

```
for row in 1..6 {
  flux line = ""
  for col in 1..6 {
    let product = row * col
    line = line + "${product}\t"
  }
  print(line)
}
// Output:
// 1  2  3  4  5
// 2  4  6  8  10
// 3  6  9  12 15
// 4  8  12 16 20
// 5  10 15 20 25
```

The compiler handles nesting by saving and restoring the loop state (break jump list, loop start address, loop depth, and local counts) at each loop boundary. This means `break` and `continue` always apply to the innermost loop, regardless of nesting depth.

5.3 break, continue, return

These three keywords give you fine-grained control over how loops and functions execute.

5.3.1 break — Exit the Loop

The `break` keyword immediately exits the innermost enclosing loop:

Listing 5.16: Finding the first negative number

```
let readings = [12, 45, -3, 67, -8, 22]

for reading in readings {
  if reading < 0 {
    print("First negative: ${reading}")
    break
  }
}
// Output: First negative: -3
```

When the compiler encounters `break`, it emits `OP_POP` instructions to clean up any locals declared inside the loop (or `OP_CLOSE_UPVALUE` if the local has been captured by a closure), then emits an `OP_JUMP` whose target is patched later to point just past the loop body. The compiler tracks break jump locations in a dynamic array, handling nested loops correctly by saving and restoring the break count at each loop boundary.

break Outside a Loop

Using `break` outside of a loop is a compile-time error. The compiler checks `loop_depth` and rejects the program with `"break outside of loop"` if you try.

5.3.2 continue — Skip to the Next Iteration

The `continue` keyword skips the rest of the current iteration and jumps to the next one:

Listing 5.17: Skipping negative readings

```
let readings = [12, -3, 45, -8, 67]

for reading in readings {
  if reading < 0 {
    continue
  }
  print("Processing: ${reading}")
}
// Output:
// Processing: 12
// Processing: 45
// Processing: 67
```

Like `break`, the compiler pops any locals declared inside the loop before jumping. But instead of jumping past the loop, `continue` emits an `OP_LOOP` that jumps back to the loop's start point—the condition check for `while` loops, or the `OP_ITER_NEXT` for `for` loops.

The compiler separately tracks the local count for `break` and `continue` because in a `for` loop they have different cleanup requirements. The `continue` path must preserve the iterator state (the two anonymous locals for the collection and index counter) while only popping the loop variable and body-local variables. The `break` path, on the other hand, pops everything including the iterator state.

Listing 5.18: continue and break working together

```
let data = [10, 0, 20, 0, 30, 0, 40]
flux sum = 0

for value in data {
  if value == 0 {
    continue  // skip zeros
  }
  sum = sum + value
  if sum > 50 {
    break  // stop once we exceed 50
  }
}

print(sum) // Output: 60 (10 + 20 + 30)
```

5.3.3 return — Exit the Function

The `return` keyword exits the current function immediately, optionally with a value:

Listing 5.19: Early return from a function

```
fn find_index(haystack: Array, needle: any) -> Int {
  for i in 0..haystack.len() {
    if haystack[i] == needle {
      return i
    }
  }
  return -1
}

let colors = ["red", "green", "blue"]
print(find_index(colors, "green"))  // Output: 1
print(find_index(colors, "yellow")) // Output: -1
```

If you omit the value after `return`, the function returns `unit`—Lattice's equivalent of "nothing meaningful." The compiler emits `OP_UNIT` in that case, followed by any return-type checks, `ensure` contract checks, and `defer` cleanup (via `OP_DEFER_RUN`) before the final `OP_RETURN` instruction.

Listing 5.20: Implicit return: no return keyword needed

```
fn double(x: Int) -> Int {
  x * 2
}

print(double(21)) // Output: 42
```

Since functions are expressions too, the last expression in a function body becomes its return value. You only need an explicit `return` for early exits.

Listing 5.21: Guard clauses with early return

```
fn process_order(order: Map) -> String {
  if order["status"] == "cancelled" {
    return "Order was cancelled"
  }
  if order["total"] == nil {
    return "Invalid order: no total"
  }
  if order["total"] < 0 {
    return "Invalid order: negative total"
  }

  // Main logic only reached if all guards pass
  let tax = order["total"] * 0.08
  let final_amount = order["total"] + tax
  "Processed: total = ${final_amount}"
}
```

This "guard clause" pattern—checking error conditions at the top and returning early—keeps the happy path at the main indentation level, making code easier to follow.

break, continue, and Closures

These keywords only affect the innermost loop in the current function. You cannot `break` out of a closure's loop from outside the closure, nor `continue` a loop that belongs to a different function. Each function has its own loop tracking.

5.4 Ranges

Ranges represent a sequence of integers from a start value to an end value. They appear frequently in `for` loops and array slicing.

5.4.1 Creating Ranges

The `..` operator creates a half-open range—inclusive of the start, exclusive of the end:

Listing 5.22: Half-open ranges

```
let countdown = 0..5
print(countdown) // Output: 0..5

for n in 0..5 {
  print(n)
}
// Output: 0 1 2 3 4
```

The range `0..5` includes 0, 1, 2, 3, and 4—but not 5. This is the same convention used by Python's `range()` and Rust's `..` operator, and it aligns naturally with zero-indexed arrays: `0..array.len()` covers every valid index.

Range

A *range* is a value of type `Range` containing a start integer and an end integer. The notation `a..b` creates a range from `a` (inclusive) to `b` (exclusive). Internally, ranges are stored as a pair of `int64_t` values in the `LatValue` union (see `include/value.h`) and require no heap allocation—they're as cheap to create as a pair of integers.

5.4.2 Ranges with Expressions

Both endpoints of a range can be arbitrary expressions, not only literals:

Listing 5.23: Dynamic range endpoints

```
let page = 2
let page_size = 10
let start = page * page_size
let end = start + page_size

for i in start..end {
  print("Item ${i}")
}
// Output: Item 20, Item 21, ... Item 29
```

The parser handles this by parsing each side of the `..` token as an addition-level expression. This means arithmetic works naturally in range endpoints, but if you need lower-precedence operators you should use parentheses.

5.4.3 Ranges for Indexing and Slicing

Ranges are not only for loops—they work as indices to slice arrays and strings:

Listing 5.24: Slicing with ranges

```
let letters = ["a", "b", "c", "d", "e"]

let middle = letters[1..4]
print(middle) // Output: ["b", "c", "d"]

let greeting = "Hello, World!"
let word = greeting[0..5]
print(word) // Output: Hello
```

The range `1..4` selects elements at indices 1, 2, and 3—three elements total. This consistency between loop ranges and slice ranges means you can reason about both the same way: the range `a..b` always means "from `a` up to but not including `b`."

5.4.4 Ranges as First-Class Values

Ranges are first-class values. You can store them in variables, pass them to functions, and return them:

Listing 5.25: Ranges as first-class values

```
fn page_range(page: Int, page_size: Int) -> Range {
  let start = page * page_size
  start..(start + page_size)
}

let items = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"]
let second_page = items[page_range(1, 3)]
print(second_page) // Output: ["d", "e", "f"]
```

At the bytecode level, the compiler emits `OP_BUILD_RANGE` which pops the end and start values from the stack and pushes a `Range` value. Since ranges contain only two integers stored inline in the value union, they're stack-allocated and extremely cheap to create—no heap allocation, no garbage collection pressure.

Listing 5.26: Using ranges for batch processing

```
fn process_batch(items: Array, batch_size: Int) -> Array {
  let batches = []
  flux offset = 0
  while offset < items.len() {
    let end = if offset + batch_size > items.len() {
      items.len()
    } else {
      offset + batch_size
    }
    batches.push(items[offset..end])
    offset = offset + batch_size
  }
  batches
}

let numbers = [1, 2, 3, 4, 5, 6, 7]
let batched = process_batch(numbers, 3)
print(batched) // Output: [[1, 2, 3], [4, 5, 6], [7]]
```

5.5 The Nil-Coalescing Operator ??

Working with optional data is a fact of life. APIs return `nil` when a key is missing, functions return `nil` on failure, and user input may be absent. The nil-coalescing operator `??` provides a clean way to supply a fallback:

Listing 5.27: Basic nil-coalescing

```
let config = Map::new()
config["theme"] = "dark"

let theme = config["theme"] ?? "light"
let language = config["language"] ?? "en"

print(theme)    // Output: dark
print(language) // Output: en
```

The expression `a ?? b` evaluates `a`. If the result is not `nil`, it becomes the value of the whole expression. If `a` is `nil`, then `b` is evaluated and used instead.

Nil-Coalescing Operator

The operator `??` evaluates its left operand first. If that value is not `nil`, the right operand is *never evaluated* (short-circuit semantics). If the left operand is `nil`, the right operand is evaluated and its value is returned.

5.5.1 Short-Circuit Evaluation

The right side of `??` is only evaluated when needed. This matters when the fallback involves computation:

Listing 5.28: Short-circuit evaluation in action

```
fn expensive_default() -> String {
  print("Computing default...")
  "fallback"
}

let cached_value = "cached"
let result = cached_value ?? expensive_default()
print(result)
// Output: cached
// (expensive_default was never called)
```

The compiler implements this with a conditional jump. It compiles the left operand, then emits `OP_JUMP_IF_NOT_NIL`—if the value on the stack is not nil, execution skips over the right operand entirely. Only if the value is nil does it pop the nil, evaluate the fallback, and continue. This means the cost of a `??` expression when the left side is non-nil is a single comparison and branch—no function call overhead, no allocation.

5.5.2 Chaining ??

You can chain multiple `??` operators to try several sources in order:

Listing 5.29: Chaining nil-coalescing operators

```
let env_port = nil
let config_port = nil
let default_port = 8080

let port = env_port ?? config_port ?? default_port
print(port) // Output: 8080
```

Each `??` short-circuits independently. If `env_port` is not `nil`, neither `config_port` nor `default_port` is evaluated. If `env_port` is `nil` but `config_port` is not, `default_port` is skipped. This creates a natural priority chain for configuration values.

Listing 5.30: A practical configuration lookup

```
fn get_setting(key: String, env: Map, config: Map) -> String {
  env[key] ?? config[key] ?? "default_${key}"
}

let env_vars = Map::new()
let config_file = Map::new()
config_file["port"] = "3000"

print(get_setting("port", env_vars, config_file))    // Output: 3000
print(get_setting("host", env_vars, config_file))     // Output: default_host
```

5.5.3 Combining ?? with Other Expressions

Because `??` is an expression, it composes with everything else:

Listing 5.31: Nil-coalescing in various contexts

```
let scores = Map::new()
scores["alice"] = 95

// In function arguments
print(scores["bob"] ?? 0) // Output: 0

// In arithmetic
let alice_score = scores["alice"] ?? 0
let bob_score = scores["bob"] ?? 0
let total = alice_score + bob_score
print(total) // Output: 95

// Combined with if/else
let grade = if (scores["alice"] ?? 0) >= 90 {
  "A"
} else {
  "B"
}
print(grade) // Output: A
```

?? vs. if/else for Defaults

Use `??` when you have a value that might be `nil` and want a fallback. Use `if`/`else` when you need to test a condition more complex than nil-ness—like checking whether a number is positive or a string is non-empty. The `??` operator only checks for `nil`; it does not treat `false`, `0`, or `""` as missing.

Constant Folding and ??

The compiler's constant-folding pass (in `src/stackcompiler.c`) deliberately skips `??` expressions. Because the operator has short-circuit semantics—the right side may have side effects that should only execute when the left is nil—the compiler does not attempt to evaluate it at compile time. The same applies to `&&` and `||`.

5.6 The Pipe Operator

When you chain multiple transformations on a value, code can become deeply nested and hard to read:

Listing 5.32: Deeply nested function calls

```
// Reading inside-out: replace, then trim, then to_upper
let result = to_upper(trim(replace(raw_input, "  ", " ")))
```

You have to read this inside-out: first `replace`, then `trim`, then `to_upper`. The `pipe()` built-in lets you write transformations in the order they happen:

Listing 5.33: Using pipe for clarity

```
let words = [3, 1, 4, 1, 5, 9, 2, 6]

let result = pipe(
  words,
  |arr| arr.filter(|x| x > 3),
  |arr| arr.map(|x| x * 10),
  |arr| arr.sort_by(|a, b| a - b)
)

print(result) // Output: [40, 50, 60, 90]
```

pipe()

`pipe(value, fn1, fn2, ...)` threads a value through a series of functions from left to right. It evaluates `fn1(value)`, passes the result to `fn2`, passes that result to `fn3`, and so on. The final function's return value is the result of the entire `pipe()` call.

5.6.1 How pipe() Works

The `pipe()` function is a variadic built-in that accepts a starting value followed by any number of closures. Each closure receives the result of the previous one:

Listing 5.34: Step-by-step pipe transformation

```
let name = pipe(
  "  hello world  ",
  |s| s.trim(),
  |s| s.replace("world", "lattice"),
  |s| s.to_upper()
)

print(name) // Output: HELLO LATTICE
```

This is equivalent to the following sequential code:

Listing 5.35: The same transformations without pipe

```
let step1 = "  hello world  "
let step2 = step1.trim()
let step3 = step2.replace("world", "lattice")
let name = step3.to_upper()

print(name) // Output: HELLO LATTICE
```

The `pipe` version is more compact, and—crucially—it makes the data flow explicit. You read top to bottom and see each transformation in sequence without introducing intermediate variable names.

5.6.2 Building Data Pipelines

`pipe()` shines when processing collections:

Listing 5.36: A data processing pipeline

```
let transactions = [
  ["deposit", 100],
  ["withdrawal", 50],
  ["deposit", 200],
  ["withdrawal", 30],
  ["deposit", 150]
]

let total_deposits = pipe(
  transactions,
  |txns| txns.filter(|t| t[0] == "deposit"),
  |txns| txns.map(|t| t[1]),
  |amounts| amounts.reduce(0, |sum, x| sum + x)
)

print(total_deposits) // Output: 450
```

Each step of the pipeline is self-contained and readable. If you need to add a step—say, filtering out deposits below 150—you insert one closure in the chain:

Listing 5.37: Adding a step to the pipeline

```
let large_deposits = pipe(
  transactions,
  |txns| txns.filter(|t| t[0] == "deposit"),
  |txns| txns.map(|t| t[1]),
  |amounts| amounts.filter(|a| a >= 150),
  |amounts| amounts.reduce(0, |sum, x| sum + x)
)

print(large_deposits) // Output: 350
```

5.6.3 pipe() with Named Functions

You're not limited to inline closures. Any function that takes a single argument works:

Listing 5.38: Using named functions with pipe

```
fn double(x: Int) -> Int { x * 2 }
fn add_one(x: Int) -> Int { x + 1 }
fn square(x: Int) -> Int { x * x }

let result = pipe(5, double, add_one, square)
print(result) // Output: 121
// 5 -> 10 -> 11 -> 121
```

Under the hood, `pipe()` is implemented as a native function registered in `src/runtime.c`. The implementation deep-clones the initial value, then iterates through each closure argument, calling it with the current value and replacing the current value with the result. If any argument is not a closure, `pipe()` returns `nil`.

Listing 5.39: Combining named functions and closures in pipe

```
fn normalize(scores: Array) -> Array {
  let max_score = scores.reduce(0, |a, b| if a > b { a } else { b })
  if max_score == 0 { return scores }
  scores.map(|s| s * 100 / max_score)
}

let raw_scores = [45, 82, 67, 91, 73]

let report = pipe(
  raw_scores,
  normalize,
  |scores| scores.sort_by(|a, b| b - a),
  |scores| scores.map(|s| "${s}%")
)

print(report) // Output: ["100%", "90%", "80%", "73%", "49%"]
```

pipe() vs. Method Chaining

When working with a single collection type, method chaining (`arr.filter(...).map(...)`) is often more concise. Use `pipe()` when you're combining transformations from different sources, when intermediate steps involve standalone functions, or when you want to make the data flow unmistakably clear.

5.7 Putting It All Together

Let's combine what we've learned into a more substantial example—a function that analyzes a list of student test scores:

Listing 5.40: Comprehensive control flow example

```
fn analyze_scores(scores: Array) -> Map {
  let results = Map::new()

  flux total = 0
  flux count = 0
  flux highest = nil
  flux lowest = nil

  for score in scores {
    if score < 0 {
      continue  // skip invalid scores
    }

    total = total + score
    count = count + 1
    highest = if highest == nil { score }
              else if score > highest { score }
              else { highest }
    lowest = if lowest == nil { score }
             else if score < lowest { score }
             else { lowest }
  }

  results["count"] = count
  results["total"] = total
  results["average"] = if count > 0 { total / count } else { 0 }
  results["highest"] = highest ?? 0
  results["lowest"] = lowest ?? 0

  // Classify grades
  flux grade_counts = Map::new()
  grade_counts["A"] = 0
  grade_counts["B"] = 0
  grade_counts["C"] = 0
  grade_counts["F"] = 0

  for score in scores {
    if score < 0 { continue }

    let grade = if score >= 90 { "A" }
                else if score >= 80 { "B" }
                else if score >= 70 { "C" }
                else { "F" }

    grade_counts[grade] = grade_counts[grade] + 1
  }

  results["grades"] = grade_counts
```

This example exercises `for` loops, `continue` to skip invalid data, `if`/`else` as expressions for computing grades and min/max values, and `??` for safe defaults. Notice how every `if`/`else` chain produces a value directly—there's no need for temporary variables or mutable accumulators beyond the ones that genuinely need to change.

5.8 Exercises

1. **FizzBuzz with Expressions.** Write a `for` loop over `1..101` that prints `"Fizz"` for multiples of 3, `"Buzz"` for multiples of 5, `"FizzBuzz"` for multiples of both, and the number itself otherwise. Use `if`/`else` as an expression to determine what to print—your `print()` call should appear only once.

2. **First Match.** Write a function `fn first_match(items: Array, predicate: any) -> any` that returns the first element for which `predicate(element)` is truthy, or `nil` if no element matches. Use a `for` loop and `return` in your solution.

3. **Config Lookup Chain.** Given three maps representing environment variables, a config file, and hardcoded defaults, write a function `fn get_config(key: String, env: Map, config: Map, defaults: Map) -> any` that looks up a key in each source using chained `??` operators.

4. **Pipeline Processing.** Given an array of strings representing log lines like `"2024-01-15 ERROR disk full"`, use `pipe()` to: (1) filter only lines containing `"ERROR"`, (2) extract the date portion (the first 10 characters of each line), and (3) collect the unique dates into a sorted array.

5. **Nested Loop Challenge.** Write a function that takes an integer `n` and returns an array of all pairs `[i, j]` where `0 <= i < j < n` and `i + j` is even. Use `continue` to skip pairs where the sum is odd, and build the result with a `flux` array.

What's Next?

We've seen how to steer the flow of a program with conditions, loops, and transformations. But all of our code so far lives at the top level or in small helper functions. In Chapter 6, we'll dive deep into Lattice's function system—defining functions with type annotations, default parameters, and variadic arguments. We'll also meet closures, one of Lattice's most powerful features, and learn how they capture their surrounding environment to create flexible, reusable abstractions.

Chapter 6

Functions and Closures

A program without functions is a monologue. It says everything once, from top to bottom, and there's no way to reuse a thought. Functions let us name ideas, parameterize them, and compose them into larger ones. Closures go further—they let functions carry context with them, remembering the environment where they were born.

In this chapter, we'll explore how Lattice defines functions, annotates their types, handles default and variadic parameters, creates closures, and even enforces contracts on function behavior. By the end, you'll have the tools to write code that is modular, testable, and expressive.

6.1 Defining Functions with fn

Here's the simplest function you can write in Lattice:

Listing 6.1: A basic function

```
fn greet(name: String) -> String {
  "Hello, ${name}!"
}

print(greet("Lattice")) // Output: Hello, Lattice!
```

The anatomy is straightforward: `fn` introduces a function, followed by its name, a parenthesized parameter list with type annotations, an optional return type after `->`, and a body in curly braces. The last expression in the body becomes the return value—no explicit `return` needed.

Function Anatomy

```
fn name(param1: Type1, param2: Type2) -> ReturnType {
  // body
  // last expression is the return value
}
```

Every parameter *must* have a type annotation. The return type is optional—if omitted, the function can return any type.

6.1.1 Functions Are Values

In Lattice, functions are first-class values. You can bind them to variables, store them in arrays, pass them as arguments, and return them from other functions:

Listing 6.2: Functions as values

```
fn add(a: Int, b: Int) -> Int { a + b }
fn multiply(a: Int, b: Int) -> Int { a * b }

let operation = add
print(operation(3, 4)) // Output: 7

let ops = [add, multiply]
for op in ops {
  print(op(5, 3))
}
// Output: 8 15
```

When the compiler encounters a function definition, it compiles the body into its own bytecode chunk and wraps it in a closure value (`VAL_CLOSURE`). Even named functions are closures under the hood—they just happen to have a name and no captured variables.

6.1.2 Function Bodies as Expressions

Because Lattice is expression-oriented (Section 5.1), a function body can be a single expression, an `if`/`else` chain, or a multi-statement block:

Listing 6.3: Various function body styles

```
// Single expression
fn square(x: Int) -> Int { x * x }

// if/else as the body
fn abs_val(x: Int) -> Int {
  if x < 0 { -x } else { x }
}

// Multi-statement with implicit return
fn hypotenuse(a: Float, b: Float) -> Float {
  let a_sq = a * a
  let b_sq = b * b
  (a_sq + b_sq)
}
```

In the last example, the variables `a_sq` and `b_sq` are local to the function body. The final expression `(a_sq + b_sq)` becomes the return value. If a function body ends with a statement rather than an expression (for instance, a `print()` call), the function returns `unit`.

6.1.3 Recursive Functions

Functions can call themselves. Here's the classic factorial:

Listing 6.4: Recursive factorial

```
fn factorial(n: Int) -> Int {
  if n <= 1 {
    1
  } else {
    n * factorial(n - 1)
  }
}

print(factorial(10)) // Output: 3628800
```

The VM has a call frame limit (`STACKVM_FRAMES_MAX`) to prevent runaway recursion from consuming all memory. If you hit it, you'll get a `"stack overflow"` error—a signal to consider an iterative approach or tail-call optimization patterns.

6.2 Type Annotations on Parameters and Return Types

Every parameter in a Lattice function must carry a type annotation:

Listing 6.5: Type annotations

```
fn format_price(amount: Float, currency: String) -> String {
  "${currency} ${amount}"
}

print(format_price(29.99, "USD")) // Output: USD 29.99
```

These annotations serve two purposes. First, they act as documentation: anyone reading the function signature knows what types to pass. Second, they're *checked at runtime*. When you call `format_price`, the VM verifies that the first argument is a `Float` and the second is a `String`.

6.2.1 How Runtime Type Checking Works

At the start of every compiled function, the compiler emits `OP_CHECK_TYPE` instructions for each annotated parameter. Each instruction carries three pieces of information: the parameter's stack slot, the expected type name, and an error message template. At runtime, the VM compares the actual value's type against the expected type. If they don't match, you get a clear error:

Listing 6.6: Type mismatch error

```
fn double(x: Int) -> Int { x * 2 }
double("hello")
// Error: function 'double' parameter 'x' expects type Int, got String
```

The any Type

If a parameter can accept any type, annotate it with `any` (or `Any`). The compiler skips the type check for that parameter entirely:

```
fn identity(x: any) -> any { x }
```

6.2.2 Return Type Checking

When you specify a return type with `->`, the compiler emits an `OP_CHECK_RETURN_TYPE` instruction just before the function returns. This catches bugs where a function promises one type but delivers another:

Listing 6.7: Return type checking

```
fn get_count(items: Array) -> Int {
  if items.len() == 0 {
    "empty"  // Oops! Returning a String
  } else {
    items.len()
  }
}
// Error: function 'get_count' return type expects Int, got String
```

The error message even includes a "did you mean?" suggestion if the type name looks like a misspelling of a known type—a quality-of-life feature you can see implemented in `src/stackvm.c`.

6.2.3 Common Type Names

Type Annotation	Matches
`Int`	Integer values
`Float`	Floating-point values
`String`	String values
`Bool`	`true` or `false`
`Array`	Array values
`Map`	Map values
`Set`	Set values
`Range`	Range values
`Fn`	Closure/function values
`any` / `Any`	Any type (no check)

Table 6.1: Built-in type annotations

Type Annotations Are Not Static Types

Lattice's type annotations are checked at *runtime*, not compile time. This gives you safety without the overhead of a full static type system. For stricter checking at compile time, see Strict Mode in Chapter 14.

6.3 Default Parameter Values and Variadic Parameters

Real-world functions often need flexibility in how they're called. Lattice provides two mechanisms: default values for optional parameters and variadic parameters for variable-length argument lists.

6.3.1 Default Parameter Values

A parameter can specify a default value using =:

Listing 6.8: Default parameter values

```
fn connect(host: String, port: Int = 8080, secure: Bool = false) -> String {
  let protocol = if secure { "https" } else { "http" }
  "${protocol}://${host}:${port}"
}

print(connect("example.com"))              // Output: http://example.com:8080
print(connect("example.com", 443))         // Output: http://example.com:443
print(connect("example.com", 443, true))   // Output: https://example.com:443
```

When you omit arguments with defaults, the VM fills them in automatically. The defaults are evaluated at *compile time* by the compiler's constant-folding evaluator (`const_eval_expr`) and stored on the function's bytecode chunk. At call time, `stackvm_adjust_call_args` pushes the default values onto the stack for any missing arguments.

Default Values Are Computed Once

Default values are evaluated when the function is compiled, not each time it's called. This means defaults should be constant expressions—literals, simple arithmetic, or string values. Don't rely on a default value being recomputed on every call.

Parameters with defaults must come after required parameters:

Listing 6.9: Required parameters before defaults

```
fn create_user(name: String, role: String = "viewer", active: Bool = true) -> Map {
  let user = Map::new()
  user["name"] = name
  user["role"] = role
  user["active"] = active
  user
}

let admin = create_user("Alice", "admin")
let viewer = create_user("Bob")

print(admin["role"])   // Output: admin
print(viewer["role"])  // Output: viewer
print(viewer["active"]) // Output: true
```

6.3.2 Variadic Parameters

A variadic parameter, marked with `...`, collects any remaining arguments into an array:

Listing 6.10: Variadic parameters

```
fn log_message(level: String, ...messages: any) -> String {
  let parts = messages.join(", ")
  "[${level}] ${parts}"
}

print(log_message("INFO", "Server started"))
// Output: [INFO] Server started

print(log_message("ERROR", "Connection failed", "retrying", "attempt 3"))
// Output: [ERROR] Connection failed, retrying, attempt 3
```

The variadic parameter must be the last parameter. When the VM processes the call, `stackvm_adjust_call_args` gathers all extra arguments beyond the required count, bundles them into an array, and pushes that array as the final argument. Inside the function body, the variadic parameter behaves like a normal `Array`.

Listing 6.11: Variadic math functions

```
fn sum(...numbers: Int) -> Int {
  numbers.reduce(0, |acc, n| acc + n)
}

fn average(...numbers: Float) -> Float {
  let total = numbers.reduce(0.0, |acc, n| acc + n)
  total / numbers.len()
}

print(sum(1, 2, 3, 4, 5))         // Output: 15
print(average(85.0, 92.0, 78.0))  // Output: 85.0
```

6.3.3 Combining Defaults and Variadics

You can use both defaults and variadics in the same function, as long as the variadic parameter comes last:

Listing 6.12: Defaults and variadics together

```
fn build_query(table: String, limit: Int = 100, ...conditions: String) -> String {
  flux query = "SELECT * FROM ${table}"
  if conditions.len() > 0 {
    let where_clause = conditions.join(" AND ")
    query = query + " WHERE ${where_clause}"
  }
  query + " LIMIT ${limit}"
}

print(build_query("users"))
// Output: SELECT * FROM users LIMIT 100

print(build_query("orders", 10, "status = 'active'", "total > 50"))
// Output: SELECT * FROM orders WHERE status = 'active' AND total > 50 LIMIT 10
```

The VM determines the minimum number of required arguments as: total parameters minus defaults minus the variadic slot. If you provide fewer arguments than this minimum, or more than the non-variadic count when there's no variadic parameter, you'll get a clear error.

6.4 Closures: |x| x * 2

A closure is an anonymous function—a function without a name. In Lattice, closures use the pipe syntax:

Listing 6.13: Basic closure syntax

```
let double = |x| x * 2
let add = |a, b| a + b

print(double(21))  // Output: 42
print(add(10, 20)) // Output: 30
```

The syntax is compact: parameters between | pipes, followed by the body. For single-expression closures, no braces are needed. For multi-line bodies, use curly braces:

Listing 6.14: Multi-line closures

```
let format_name = |first, last| {
  let full = "${first} ${last}"
  full.to_upper()
}

print(format_name("jane", "doe")) // Output: JANE DOE
```

Closure Syntax

```
// Single expression
|params| expression

// Multi-statement body
|params| { statements; last_expression }

// No parameters
|| { body }
```

Closures in Lattice do not require type annotations on their parameters, unlike named functions. This keeps them lightweight for use in callbacks, iterators, and higher-order functions.

6.4.1 Closures with Default and Variadic Parameters

Closures support the same default and variadic features as named functions:

Listing 6.15: Closures with defaults and variadics

```
let greet = |name, greeting = "Hello"| "${greeting}, ${name}!"
print(greet("World"))              // Output: Hello, World!
print(greet("World", "Howdy"))     // Output: Howdy, World!

let join_all = |separator, ...parts| parts.join(separator)
print(join_all("-", "2024", "01", "15")) // Output: 2024-01-15
```

6.4.2 Where Closures Shine

Closures are the workhorses of Lattice's collection methods. You've already seen them with `filter`, `map`, and `reduce` in Chapter 5. They're used anywhere you need a short, inline function:

Listing 6.16: Closures in collection methods

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

let evens = numbers.filter(|n| n % 2 == 0)
print(evens) // Output: [2, 4, 6, 8, 10]

let doubled = numbers.map(|n| n * 2)
print(doubled) // Output: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

let sum = numbers.reduce(0, |acc, n| acc + n)
print(sum) // Output: 55

let sorted = ["banana", "apple", "cherry"].sort_by(|a, b| {
  if a < b { -1 } else if a > b { 1 } else { 0 }
})
print(sorted) // Output: ["apple", "banana", "cherry"]
```

6.5 Closures Capture Their Environment

What makes closures special is not that they're anonymous—it's that they *remember* the environment where they were created. Variables from an enclosing scope that a closure references are called *upvalues*:

Listing 6.17: Closures capture enclosing variables

```
fn make_counter() -> Fn {
  flux count = 0
  || {
    count = count + 1
    count
  }
}

let counter = make_counter()
print(counter()) // Output: 1
print(counter()) // Output: 2
print(counter()) // Output: 3
```

The closure created inside `make_counter` captures the `flux` `count` variable. Even after `make_counter` has returned, the closure retains a reference to `count` and can modify it. Each call to `make_counter` creates a new, independent counter.

Upvalue

An *upvalue* is a reference to a variable in an enclosing function's scope that a closure captures. When the enclosing function returns, the VM "closes" the upvalue—copying the variable's current value to heap storage so the closure can continue to access it.

6.5.1 How Upvalue Capture Works Under the Hood

The implementation involves three stages, visible in `src/stackcompiler.c` and `src/stackvm.c`:

1. **Compile time:** When the compiler sees a variable reference that doesn't resolve in the current function, it walks up the enclosing compiler chain via `resolve_upvalue`. If the variable is a local in the parent function, it marks that local as "captured" and records it with `is_local = true`. If it's an upvalue of the parent, it chains through with `is_local = false`. The upvalue descriptors are emitted after the `OP_CLOSURE` instruction.

2. **Closure creation (runtime):** When the VM executes `OP_CLOSURE`, it reads the upvalue descriptors. For each `is_local = true` entry, it calls `capture_upvalue` which creates an `ObjUpvalue` pointing at the live stack slot. For `is_local = false`, it reuses the parent frame's upvalue. The VM maintains a linked list of open upvalues, sorted by stack location, to ensure that multiple closures capturing the same variable share the same `ObjUpvalue`.

3. **Closing (when the scope exits):** When a function returns or a scope ends, the VM calls `close_upvalues`, which walks the open upvalue list. For each upvalue whose location is about to be popped from the stack, it deep-clones the value into the `ObjUpvalue`'s `closed` field and redirects the pointer. The closure now reads from heap storage instead of the stack.

This mechanism ensures that closures see the *latest* value of captured variables (not a snapshot from creation time), and that modifications through one closure are visible to others sharing the same upvalue.

Listing 6.18: Multiple closures sharing an upvalue

```
fn make_pair() -> Array {
  flux value = 0
  let getter = || value
  let setter = |v| { value = v }
  [getter, setter]
}

let pair = make_pair()
let get = pair[0]
let set = pair[1]

print(get()) // Output: 0
set(42)
print(get()) // Output: 42
```

Both `getter` and `setter` capture the same `flux value` variable. When `set(42)` modifies it, the change is visible through `get()`.

6.5.2 Closures in Loops

A common pitfall in other languages is creating closures inside loops and finding they all share the same variable. Lattice avoids this because `for` loop variables are fresh bindings on each iteration:

Listing 6.19: Closures in loops capture correctly

```
let multipliers = []
for i in 1..5 {
  multipliers.push(|x| x * i)
}

print(multipliers[0](10)) // Output: 10  (1 * 10)
print(multipliers[1](10)) // Output: 20  (2 * 10)
print(multipliers[2](10)) // Output: 30  (3 * 10)
print(multipliers[3](10)) // Output: 40  (4 * 10)
```

Each closure captures its own `i`. The compiler's upvalue handling ensures that when the loop variable is popped at the end of each iteration, any closures that captured it get their own closed-over copy.

Closures over flux Variables in while Loops

In a `while` loop, a `flux` counter variable is *not* rebound each iteration—it's the same mutable slot throughout. Closures created in such loops will all share the same variable. If you need per-iteration capture in a `while` loop, create a `let` binding inside the loop body:

```
flux i = 0
while i < 5 {
  let captured_i = i  // fresh binding each iteration
  callbacks.push(|x| x * captured_i)
  i = i + 1
}
```

6.6 Higher-Order Functions

A higher-order function is one that takes a function as an argument or returns a function as its result. We've seen the "takes a function" pattern extensively with collection methods. Now let's explore the "returns a function" pattern.

6.6.1 Functions That Return Functions

Listing 6.20: A function factory

```
fn make_multiplier(factor: Int) -> Fn {
  |x| x * factor
}

let triple = make_multiplier(3)
let times_ten = make_multiplier(10)

print(triple(7))      // Output: 21
print(times_ten(7))   // Output: 70
```

Each call to `make_multiplier` produces a new closure that remembers its `factor`. This pattern—a function that creates specialized functions—is sometimes called a *function factory*.

6.6.2 Callbacks and Event Handlers

Higher-order functions are natural for callback patterns:

Listing 6.21: Processing with callbacks

```
fn process_items(items: Array, on_success: Fn, on_error: Fn) -> Array {
  flux results = []
  for item in items {
    if item > 0 {
      results.push(on_success(item))
    } else {
      results.push(on_error(item))
    }
  }
  results
}

let data = [10, -3, 25, -7, 42]
let processed = process_items(
  data,
  |x| "OK: ${x}",
  |x| "ERR: invalid value ${x}"
)

print(processed)
// Output: ["OK: 10", "ERR: invalid value -3", "OK: 25",
//          "ERR: invalid value -7", "OK: 42"]
```

6.6.3 Function Composition

Lattice provides a built-in `compose` function for right-to-left composition:

Listing 6.22: Composing functions

```
fn add_one(x: Int) -> Int { x + 1 }
fn double(x: Int) -> Int { x * 2 }

// compose(f, g) returns a function where f(g(x))
let double_then_add = compose(add_one, double)
print(double_then_add(5)) // Output: 11 (double(5)=10, add_one(10)=11)

let add_then_double = compose(double, add_one)
print(add_then_double(5)) // Output: 12 (add_one(5)=6, double(6)=12)
```

For left-to-right composition (which often reads more naturally), use `pipe()` from Section 5.6:

Listing 6.23: pipe for left-to-right composition

```
let result = pipe(5, add_one, double)
print(result) // Output: 12 (add_one(5)=6, double(6)=12)
```

6.6.4 Building Abstractions with Higher-Order Functions

Higher-order functions let you abstract over patterns, not just values:

Listing 6.24: Abstracting the retry pattern

```
fn with_retry(action: Fn, max_attempts: Int = 3) -> any {
  flux attempt = 1
  loop {
    let result = try {
      action()
    } catch err {
      if attempt >= max_attempts {
        return "Failed after ${max_attempts} attempts: ${err}"
      }
      attempt = attempt + 1
      continue
    }
    return result
  }
}

// Use it to wrap any fallible operation
let data = with_retry(|| {
  // Some operation that might fail
  42
})
print(data) // Output: 42
```

Listing 6.25: A memoization wrapper

```
fn memoize(func: Fn) -> Fn {
  flux cache = Map::new()
  |arg| {
    let key = to_string(arg)
    if cache.has(key) {
      cache[key]
    } else {
      let result = func(arg)
      cache[key] = result
      result
    }
  }
}

fn slow_square(x: Int) -> Int {
  print("Computing ${x}^2...")
  x * x
}

let fast_square = memoize(slow_square)
print(fast_square(5)) // Output: Computing 5^2... \n 25
print(fast_square(5)) // Output: 25 (cached, no "Computing" message)
print(fast_square(3)) // Output: Computing 3^2... \n 9
```

The `memoize` function creates a closure that wraps the original function with a cache. The cache is a captured `flux` variable shared across all calls to the memoized function.

6.7 require and ensure — Contracts on Your Functions

Lattice supports *design by contract*: you can attach preconditions (`require`) and postconditions (`ensure`) to functions. These are runtime checks that guard against incorrect usage and document the function's expectations.

6.7.1 Preconditions with require

A `require` clause checks a condition at the *start* of the function, before the body executes:

Listing 6.26: require preconditions

```
fn withdraw(balance: Float, amount: Float) -> Float
  require amount > 0, "withdrawal amount must be positive"
  require amount <= balance, "insufficient funds"
{
  balance - amount
}

print(withdraw(100.0, 30.0)) // Output: 70.0
withdraw(100.0, -5.0)
// Error: require failed in 'withdraw': withdrawal amount must be positive
```

The syntax places `require` clauses between the function signature and the opening brace. Each clause has a boolean condition and an optional error message string. If the condition is false, the function throws an error immediately.

Under the hood, the compiler translates each `require` into a conditional: compile the condition expression, emit `OP_JUMP_IF_TRUE` to skip the error path, then emit the error message as an `OP_THROW`. This happens *after* parameter type checks but *before* the function body.

6.7.2 Postconditions with ensure

An `ensure` clause checks a condition about the *return value* after the body executes:

Listing 6.27: ensure postconditions

```
fn abs_val(x: Float) -> Float
  ensure |result| result >= 0.0, "absolute value must be non-negative"
{
  if x < 0.0 { -x } else { x }
}

print(abs_val(-42.0)) // Output: 42.0
print(abs_val(7.0))   // Output: 7.0
```

The `ensure` clause takes a *closure* that receives the return value. The closure must return a truthy value for the check to pass. This happens right before the function returns—the compiler emits the ensure check after the body but before `OP_RETURN`, so it applies to both implicit and explicit returns.

Listing 6.28: Multiple contracts on a function

```
fn divide(numerator: Float, denominator: Float) -> Float
  require denominator != 0.0, "cannot divide by zero"
  require numerator >= 0.0, "numerator must be non-negative"
  ensure |r| r >= 0.0, "result must be non-negative"
{
  numerator / denominator
}

print(divide(10.0, 3.0))  // Output: 3.3333...
divide(10.0, 0.0)
// Error: require failed in 'divide': cannot divide by zero
```

6.7.3 When to Use Contracts

Contracts work best as:

- **Guards on public API boundaries.** Validate inputs at the entry points of modules and libraries.
- **Invariant documentation.** Make implicit assumptions explicit: "this function assumes a positive input" becomes a `require` that both documents and enforces the constraint.
- **Safety nets during development.** Catch logic errors early—if an `ensure` fails, you know the function's implementation has a bug.

Contracts vs. Error Handling

Contracts check for *programmer errors*—conditions that should never happen if the code is correct. For *expected failures* (file not found, network timeout, invalid user input), use Lattice's error handling mechanisms from Chapter 10.

The contract system is compiled into the same bytecode as normal error handling: `require` compiles to a conditional `OP_THROW`, and `ensure` compiles to a closure call whose result is checked with `OP_JUMP_IF_TRUE`. You can see the detailed logic in `emit_ensure_checks` and `compile_function_body` in `src/stackcompiler.c`. There's no runtime overhead when a contract passes beyond a comparison and a branch.

6.8 Putting It All Together

Let's build something more substantial: a function pipeline builder that lets you chain operations with validation.

Listing 6.29: A validated pipeline builder

```
fn create_pipeline(...transforms: Fn) -> Fn {
  |input| {
    flux current = input
    for transform in transforms {
      current = transform(current)
    }
    current
  }
}

fn clamp(min_val: Float, max_val: Float) -> Fn
  require min_val <= max_val, "min must be <= max"
{
  |x| {
    if x < min_val { min_val }
    else if x > max_val { max_val }
    else { x }
  }
}

fn scale(factor: Float) -> Fn
  require factor != 0.0, "scale factor cannot be zero"
{
  |x| x * factor
}

fn offset(amount: Float) -> Fn {
  |x| x + amount
}

// Build a temperature converter: Celsius to Fahrenheit, clamped to 32-212
let celsius_to_fahrenheit = create_pipeline(
  scale(9.0 / 5.0),
  offset(32.0),
  clamp(32.0, 212.0)
)

let temperatures = [0.0, 25.0, 100.0, 150.0, -40.0]
for temp in temperatures {
  let converted = celsius_to_fahrenheit(temp)
  print("${temp}C = ${converted}F")
}
// Output:
// 0.0C = 32.0F
// 25.0C = 77.0F
// 100.0C = 212.0F
// 150.0C = 212.0F
// -40.0C = 32.0F
```

This example combines closures (returned by `clamp`, `scale`, `offset`), variadic parameters (in `create_pipeline`), upvalue capture (each returned closure remembers its parameters), and contracts (validating inputs to `clamp` and `scale`).

6.9 Exercises

1. **once().** Write a function `fn once(f: Fn) -> Fn` that returns a wrapper which calls `f` on the first invocation and returns the cached result on subsequent calls.
2. **Partial Application.** Write a function `fn partial(f: Fn, first_arg: any) -> Fn` that returns a new function with the first argument pre-filled. For example, `partial(add, 5)(3)` should return `8`.
3. **Validated List.** Write a function `fn make_list(validator: Fn) -> Map` that returns a map with `"add"` and `"items"` closures. The `"add"` closure should only add items that pass the validator. Use upvalue capture to share state between the closures.
4. **Function Pipeline with Error Handling.** Extend the pipeline pattern by writing `fn safe_pipeline(...transforms: Fn) -> Fn` that wraps each transform in a try/catch. If any transform fails, the pipeline should return the error message instead of crashing.
5. **Contract Challenge.** Write a function `fn binary_search` that searches a sorted array for a value. Add `require` contracts to verify the array is sorted and the target is of the correct type. Add an `ensure` contract to verify that if the function returns a non-negative index, the element at that index equals the target.

What's Next?

With functions and closures in our toolkit, we're ready to work with Lattice's rich collection types. In Chapter 7, we'll explore arrays, maps, sets, and tuples in depth—including the closure-based methods like `map`, `filter`, and `reduce` that make collections so powerful, and the spread operator that makes working with them a breeze.

Chapter 7

Arrays, Maps, Sets, and Friends

Every program of any real size works with collections of data—lists of users, tables of configuration, sets of unique tags. Lattice provides four core collection types: **arrays** for ordered sequences, **maps** for key-value associations, **sets** for unique elements, and **tuples** for fixed-length heterogeneous groups.

What makes these collections a joy to use is Lattice's rich method vocabulary. Arrays alone carry over two dozen methods, from the essential `push` and `pop` to powerful closure-based transforms like `map`, `filter`, and `reduce`. Let's explore each collection type from creation through advanced usage.

7.1 Arrays: Creation, Indexing, and Mutation

An array is an ordered, growable sequence of values. Creating one is straightforward:

Listing 7.1: Creating arrays

```
let numbers = [1, 2, 3, 4, 5]
let mixed = [42, "hello", true, 3.14]
let empty = []

print(numbers) // Output: [1, 2, 3, 4, 5]
print(mixed)   // Output: [42, "hello", true, 3.14]
print(empty)   // Output: []
```

Arrays in Lattice are heterogeneous—a single array can hold values of different types. Internally (see `include/value.h`), an array stores a pointer to a contiguous buffer of `LatValue` elements along with the current length and capacity, growing as needed.

7.1.1 Indexing and Slicing

Elements are accessed by zero-based index:

Listing 7.2: Array indexing

```
let colors = ["red", "green", "blue", "yellow"]

print(colors[0])  // Output: red
print(colors[2])  // Output: blue

// Negative indices count from the end
print(colors[-1]) // Output: yellow
print(colors[-2]) // Output: blue
```

Slicing with ranges (Section 5.4.3) extracts sub-arrays:

Listing 7.3: Array slicing

```
let letters = ["a", "b", "c", "d", "e", "f"]

let first_three = letters[0..3]
print(first_three) // Output: ["a", "b", "c"]

let middle = letters[2..5]
print(middle) // Output: ["c", "d", "e"]
```

7.1.2 Mutation with push and pop

Arrays can be modified in place when they're in a mutable phase:

Listing 7.4: push and pop

```
let fruits = ["apple", "banana"]

fruits.push("cherry")
print(fruits) // Output: ["apple", "banana", "cherry"]

let removed = fruits.pop()
print(removed) // Output: cherry
print(fruits)  // Output: ["apple", "banana"]
```

The `push` method appends an element to the end, and `pop` removes and returns the last element (or `nil` if the array is empty). Both methods respect phase constraints—if an array is in the crystal phase (`fix`), attempting to push or pop will produce an error.

Listing 7.5: Building arrays with push

```
flux primes = []
for n in 2..50 {
  flux is_prime = true
  for divisor in 2..n {
    if n % divisor == 0 {
      is_prime = false
      break
    }
  }
  if is_prime {
    primes.push(n)
  }
}
print(primes) // Output: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

7.1.3 Essential Non-Closure Methods

Here are the methods that don't take closures—the bread and butter of array manipulation:

Listing 7.6: Array inspection methods

```
let items = [10, 20, 30, 20, 40]

print(items.len())          // Output: 5
print(items.contains(20))   // Output: true
print(items.contains(99))   // Output: false
print(items.index_of(30))   // Output: 2
print(items.index_of(99))   // Output: -1
print(items.first())        // Output: 10
print(items.last())         // Output: 40
```

Listing 7.7: Array transformation methods

```
let numbers = [3, 1, 4, 1, 5, 9, 2, 6]

print(numbers.reverse())  // Output: [6, 2, 9, 5, 1, 4, 1, 3]
print(numbers.unique())   // Output: [3, 1, 4, 5, 9, 2, 6]
print(numbers.sort())     // Output: [1, 1, 2, 3, 4, 5, 6, 9]
print(numbers.sum())      // Output: 31
print(numbers.min())      // Output: 1
print(numbers.max())      // Output: 9
```

Listing 7.8: Array subsetting methods

```
let alphabet = ["a", "b", "c", "d", "e", "f", "g"]

print(alphabet.take(3))   // Output: ["a", "b", "c"]
print(alphabet.drop(4))   // Output: ["e", "f", "g"]
print(alphabet.chunk(3))  // Output: [["a", "b", "c"], ["d", "e", "f"], ["g"]]
```

Listing 7.9: join, enumerate, and zip

```
let words = ["Lattice", "is", "great"]
print(words.join(" "))  // Output: Lattice is great
print(words.join("-"))  // Output: Lattice-is-great

let items = ["apple", "banana", "cherry"]
print(items.enumerate())
// Output: [[0, "apple"], [1, "banana"], [2, "cherry"]]

let keys = ["name", "age", "city"]
let vals = ["Alice", 30, "Paris"]
print(keys.zip(vals))
// Output: [["name", "Alice"], ["age", 30], ["city", "Paris"]]
```

sort() vs. sort_by()

The `sort()` method (no closure) works only with homogeneous arrays of `Int`, `Float`, or `String`. It uses the natural ordering (ascending numeric, lexicographic for strings). For custom sorting logic or heterogeneous data, use `sort_by()` with a comparator closure.

7.2 Closure-Based Array Methods

The real power of Lattice's arrays comes from methods that accept closures. These let you express transformations declaratively—saying *what* you want rather than *how* to compute it.

7.2.1 map — Transform Every Element

`map` applies a closure to each element and collects the results:

Listing 7.10: Mapping over arrays

```
let prices = [10.0, 25.0, 7.5, 42.0]

let with_tax = prices.map(|p| p * 1.08)
print(with_tax) // Output: [10.8, 27.0, 8.1, 45.36]

let labels = prices.map(|p| "$${p}")
print(labels) // Output: ["$10.0", "$25.0", "$7.5", "$42.0"]
```

The original array is untouched—`map` always returns a new array.

7.2.2 filter — Keep Matching Elements

`filter` returns a new array containing only elements for which the closure returns `true`:

Listing 7.11: Filtering arrays

```
let temperatures = [36.5, 37.2, 38.8, 36.9, 39.1, 37.0]

let fevers = temperatures.filter(|t| t > 37.5)
print(fevers) // Output: [38.8, 39.1]

let words = ["hello", "a", "Lattice", "is", "wonderful"]
let long_words = words.filter(|w| w.len() > 3)
print(long_words) // Output: ["hello", "Lattice", "wonderful"]
```

7.2.3 reduce — Collapse to a Single Value

`reduce` folds an array into a single value using an accumulator:

Listing 7.12: Reducing arrays

```
let numbers = [1, 2, 3, 4, 5]

// Sum with explicit initial value
let sum = numbers.reduce(0, |acc, n| acc + n)
print(sum) // Output: 15

// Product
let product = numbers.reduce(1, |acc, n| acc * n)
print(product) // Output: 120

// Build a string
let words = ["the", "quick", "brown", "fox"]
let sentence = words.reduce("", |acc, w| {
  if acc == "" { w } else { "${acc} ${w}" }
})
print(sentence) // Output: the quick brown fox
```

The first argument to `reduce` is the initial accumulator value. The closure receives two arguments: the current accumulator and the next element. If you omit the initial value, the first element becomes the accumulator and iteration starts from the second.

7.2.4 sort_by — Custom Sorting

`sort_by` sorts an array using a comparator closure. The closure takes two elements and should return a negative number if the first should come before the second, positive if after, and zero if equal:

Listing 7.13: Custom sorting

```
let people = [
  ["Alice", 30],
  ["Charlie", 25],
  ["Bob", 35]
]

// Sort by age (second element)
let by_age = people.sort_by(|a, b| a[1] - b[1])
print(by_age)
// Output: [["Charlie", 25], ["Alice", 30], ["Bob", 35]]

// Sort by name length, then alphabetically
let names = ["Alice", "Bob", "Charlie", "Jo", "Eve"]
let sorted = names.sort_by(|a, b| {
  let diff = a.len() - b.len()
  if diff != 0 { diff }
  else if a < b { -1 }
  else if a > b { 1 }
  else { 0 }
})
print(sorted) // Output: ["Bo", "Jo", "Eve", "Alice", "Charlie"]  -- wait
```

7.2.5 group_by — Categorize Elements

`group_by` partitions an array into a map based on a key function:

Listing 7.14: Grouping elements

```
let words = ["apple", "avocado", "banana", "blueberry", "cherry", "coconut"]

let by_first_letter = words.group_by(|w| w[0..1])
print(by_first_letter)
// Output: {"a": ["apple", "avocado"], "b": ["banana", "blueberry"],
//          "c": ["cherry", "coconut"]}

let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let parity = numbers.group_by(|n| if n % 2 == 0 { "even" } else { "odd" })
print(parity)
// Output: {"odd": [1, 3, 5, 7, 9], "even": [2, 4, 6, 8, 10]}
```

7.2.6 More Closure Methods

Listing 7.15: find, any, all

```
let inventory = [
  ["widget", 150],
  ["gadget", 0],
  ["doohickey", 42],
  ["thingamajig", 7]
]

// find: first matching element
let out_of_stock = inventory.find(|item| item[1] == 0)
print(out_of_stock) // Output: ["gadget", 0]

// find_index: index of first match
let idx = inventory.find_index(|item| item[0] == "doohickey")
print(idx) // Output: 2

// any: at least one match?
let has_low_stock = inventory.any(|item| item[1] < 10)
print(has_low_stock) // Output: true

// all: every element matches?
let all_in_stock = inventory.all(|item| item[1] > 0)
print(all_in_stock) // Output: false
```

Both `any` and `all` short-circuit: `any` stops at the first `true`, and `all` stops at the first `false`.

Listing 7.16: flat_map, each, and partition

```
// flat_map: map then flatten one level
let sentences = ["hello world", "good morning"]
let words = sentences.flat_map(|s| s.split(" "))
print(words) // Output: ["hello", "world", "good", "morning"]

// each: side effects only (returns unit)
[1, 2, 3].each(|n| print("Item: ${n}"))
// Output: Item: 1 \n Item: 2 \n Item: 3

// partition: split into [matching, non-matching]
let nums = [1, 2, 3, 4, 5, 6, 7, 8]
let result = nums.partition(|n| n % 2 == 0)
print(result) // Output: [[2, 4, 6, 8], [1, 3, 5, 7]]
```

Chaining Methods

You can chain closure methods to build expressive data transformations:

```
let result = data
  .filter(|x| x > 0)
  .map(|x| x * 2)
  .sort_by(|a, b| a - b)
  .take(5)
```

Each method returns a new array, so the chain flows naturally from left to right.

7.3 Non-Closure Array Methods Reference

7.4 The Spread Operator . . .

The spread operator `...` unpacks an array into another array:

Method	Returns	Description
`.len()`	`Int`	Number of elements
`.contains(v)`	`Bool`	Whether array contains `v`
`.index_of(v)`	`Int`	Index of first `v`, or `-1`
`.first()`	`Any`	First element (or `unit`)
`.last()`	`Any`	Last element (or `unit`)
`.reverse()`	`Array`	New reversed array
`.unique()`	`Array`	New array without duplicates
`.sort()`	`Array`	New sorted array (natural order)
`.flatten()`	`Array`	Flatten one level of nesting
`.join(sep)`	`String`	Elements joined by separator
`.zip(other)`	`Array`	Pairs with another array
`.enumerate()`	`Array`	`[index, value]` pairs
`.take(n)`	`Array`	First `n` elements
`.drop(n)`	`Array`	Skip first `n` elements
`.chunk(n)`	`Array`	Sub-arrays of size `n`
`.sum()`	`Number`	Sum of numeric elements
`.min()`	`Number`	Smallest element
`.max()`	`Number`	Largest element
`.push(v)`	`Unit`	Append element (mutates)
`.pop()`	`Any`	Remove and return last (mutates)

Table 7.1: Non-closure array methods

Listing 7.17: Spread operator basics

```
let front = [1, 2, 3]
let back = [7, 8, 9]

let combined = [...front, 4, 5, 6, ...back]
print(combined) // Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Under the hood, when the compiler sees a spread expression (`EXPR_SPREAD`) inside an array literal, it emits the elements normally, builds the array with `OP_BUILD_ARRAY`, and then emits `OP_ARRAY_FLATTEN` to flatten one level of nesting. This means spread elements are expanded into the containing array.

Listing 7.18: Common spread patterns

```
// Prepending and appending
let items = [2, 3, 4]
let with_bookends = [1, ...items, 5]
print(with_bookends) // Output: [1, 2, 3, 4, 5]

// Merging multiple arrays
let a = [1, 2]
let b = [3, 4]
let c = [5, 6]
let merged = [...a, ...b, ...c]
print(merged) // Output: [1, 2, 3, 4, 5, 6]

// Creating a copy with modifications
let original = ["red", "green", "blue"]
let extended = [...original, "yellow", "purple"]
print(extended) // Output: ["red", "green", "blue", "yellow", "purple"]
```

Spread Only Works in Array Literals

The spread operator `...` works inside array literal brackets `[...]`. It is not a general-purpose operator that can be used in arbitrary expressions. For function arguments, the variadic `...` syntax has a different meaning (see Section 6.3.2).

7.5 Maps

A map (also known as a dictionary or hash table) stores key-value pairs. In Lattice, maps are created with `Map::new()` and populated via index assignment:

Listing 7.19: Creating and populating maps

```
let user = Map::new()
user["name"] = "Alice"
user["age"] = 30
user["email"] = "alice@example.com"

print(user["name"])  // Output: Alice
print(user["age"])   // Output: 30
```

Map Keys

Map keys in Lattice are strings internally. When you use a non-string value as a key, it's converted to its string representation. This means `m[42]` and `m["42"]` refer to the same entry.

7.5.1 Map Methods

Listing 7.20: Core map methods

```
let config = Map::new()
config["host"] = "localhost"
config["port"] = 8080
config["debug"] = true

// Checking keys
print(config.has("host"))     // Output: true
print(config.has("timeout"))  // Output: false

// Safe access with .get()
print(config.get("host"))     // Output: localhost
print(config.get("timeout"))  // Output: nil

// Size
print(config.len()) // Output: 3
```

Listing 7.21: keys, values, and entries

```
let scores = Map::new()
scores["Alice"] = 95
scores["Bob"] = 82
scores["Charlie"] = 91

print(scores.keys())    // Output: ["Alice", "Bob", "Charlie"]
print(scores.values())  // Output: [95, 82, 91]
print(scores.entries())
// Output: [["Alice", 95], ["Bob", 82], ["Charlie", 91]]
```

Listing 7.22: Removing keys and merging maps

```
let defaults = Map::new()
defaults["theme"] = "light"
defaults["language"] = "en"
defaults["font_size"] = 14

let user_prefs = Map::new()
user_prefs["theme"] = "dark"
user_prefs["font_size"] = 16

// merge: user preferences override defaults
defaults.merge(user_prefs)
print(defaults["theme"])     // Output: dark
print(defaults["language"])  // Output: en
print(defaults["font_size"]) // Output: 16

// remove
defaults.remove("language")
print(defaults.has("language")) // Output: false
```

7.5.2 Iterating Over Maps

You can iterate over a map's entries, keys, or values:

Listing 7.23: Iterating over maps

```
let inventory = Map::new()
inventory["apples"] = 50
inventory["oranges"] = 30
inventory["bananas"] = 45

// Iterate over entries
for entry in inventory.entries() {
  let item = entry[0]
  let count = entry[1]
  print("${item}: ${count} in stock")
}

// Iterate over keys
for key in inventory.keys() {
  print("We carry: ${key}")
}
```

7.5.3 Maps as Lightweight Objects

Until you learn about structs (Chapter 15), maps serve as a convenient way to group related data:

Listing 7.24: Maps as ad-hoc objects

```
fn create_point(x: Float, y: Float) -> Map {
  let point = Map::new()
  point["x"] = x
  point["y"] = y
  point
}

fn distance(p1: Map, p2: Map) -> Float {
  let dx = p1["x"] - p2["x"]
  let dy = p1["y"] - p2["y"]
  (dx * dx + dy * dy)
}

let origin = create_point(0.0, 0.0)
let target = create_point(3.0, 4.0)
print(distance(origin, target)) // Output: 25.0
```

Method	Returns	Description
`.len()`	`Int`	Number of key-value pairs
`.has(key)`	`Bool`	Whether key exists
`.get(key)`	`Any`	Value for key, or `nil`
`.keys()`	`Array`	All keys as array
`.values()`	`Array`	All values as array
`.entries()`	`Array`	`[key, value]` pairs
`.remove(key)`	`Unit`	Remove key-value pair
`.merge(other)`	`Unit`	Merge another map (mutates)

Table 7.2: Map methods

7.6 Sets

A set is an unordered collection of unique elements. Lattice sets are created from arrays using `Set::new()`:

Listing 7.25: Creating sets

```
let colors = Set::new()
colors.add("red")
colors.add("green")
colors.add("blue")
colors.add("red")  // duplicate, ignored

print(colors)      // Output: {red, green, blue}
print(colors.len()) // Output: 3
```

7.6.1 Set Operations

Sets support the classic mathematical operations:

Listing 7.26: Set operations

```
let frontend = Set::new()
frontend.add("Alice")
frontend.add("Bob")
frontend.add("Charlie")

let backend = Set::new()
backend.add("Bob")
backend.add("Diana")
backend.add("Eve")

// Union: all members of either team
let all_devs = frontend.union(backend)
print(all_devs) // Output: {Alice, Bob, Charlie, Diana, Eve}

// Intersection: members of both teams
let fullstack = frontend.intersection(backend)
print(fullstack) // Output: {Bob}

// Difference: frontend-only developers
let frontend_only = frontend.difference(backend)
print(frontend_only) // Output: {Alice, Charlie}

// Symmetric difference: in one team but not both
let specialists = frontend.symmetric_difference(backend)
print(specialists) // Output: {Alice, Charlie, Diana, Eve}
```

7.6.2 Subset and Superset Testing

Listing 7.27: Subset and superset testing

```
let required_skills = Set::new()
required_skills.add("Python")
required_skills.add("SQL")

let candidate_skills = Set::new()
candidate_skills.add("Python")
candidate_skills.add("SQL")
candidate_skills.add("JavaScript")
candidate_skills.add("Docker")

print(required_skills.is_subset(candidate_skills)) // Output: true
print(candidate_skills.is_superset(required_skills)) // Output: true
```

7.6.3 Converting Between Sets and Arrays

Listing 7.28: Set conversion

```
let duplicates = [1, 2, 2, 3, 3, 3, 4]

// Quick deduplication via Set
let unique_set = Set::new()
for item in duplicates {
  unique_set.add(item)
}
let unique_array = unique_set.to_array()
print(unique_array) // Output: [1, 2, 3, 4]
```

Internally, sets are implemented as maps where the keys are the string representations of elements and the values are the actual `LatValue` objects. This means set membership testing is O(1) on average, just like map key lookups.

7.7 Tuples

A tuple is a fixed-length, heterogeneous collection. Unlike arrays, tuples cannot grow or shrink after creation:

Method	Returns	Description
`.len()`	`Int`	Number of elements
`.has(v)`	`Bool`	Whether v is in the set
`.add(v)`	`Unit`	Add element (mutates)
`.remove(v)`	`Unit`	Remove element (mutates)
`.clear()`	`Unit`	Remove all elements (mutates)
`.to_array()`	`Array`	Convert to array
`.union(other)`	`Set`	Elements in either set
`.intersection(other)`	`Set`	Elements in both sets
`.difference(other)`	`Set`	Elements in self but not other
`.symmetric_difference(other)`	`Set`	Elements in one but not both
`.is_subset(other)`	`Bool`	All elements in other?
`.is_superset(other)`	`Bool`	Contains all of other?

Table 7.3: Set methods

Listing 7.29: Creating tuples

```
let point = (3.0, 4.0)
let record = ("Alice", 30, true)
let singleton = (42,)

print(point)   // Output: (3.0, 4.0)
print(record)  // Output: ("Alice", 30, true)
```

Note the trailing comma in `(42,)`—this distinguishes a one-element tuple from a parenthesized expression.

7.7.1 Accessing Tuple Elements

Tuple elements are accessed by index:

Listing 7.30: Accessing tuple elements

```
let person = ("Alice", 30, "alice@example.com")

print(person[0])   // Output: Alice
print(person[1])   // Output: 30
print(person[2])   // Output: alice@example.com
print(person.len()) // Output: 3
```

7.7.2 Tuples vs. Arrays

When should you use a tuple instead of an array?

- Use **tuples** for fixed collections where the position carries meaning: coordinates `(x, y)`, key-value pairs `(key, value)`, function results that return multiple values.
- Use **arrays** for variable-length collections of similar items: a list of names, a sequence of measurements, a batch of records.

Listing 7.31: Tuples for multiple return values

```
fn min_max(numbers: Array) -> Tuple {
  flux lo = numbers[0]
  flux hi = numbers[0]
  for n in numbers {
    if n < lo { lo = n }
    if n > hi { hi = n }
  }
  (lo, hi)
}

let result = min_max([38, 12, 45, 7, 91, 23])
print("Min: ${result[0]}, Max: ${result[1]}")
// Output: Min: 7, Max: 91
```

Tuples are immutable by nature. You cannot push, pop, or reassign individual elements. If you need to "update" a tuple, create a new one.

Listing 7.32: Tuples are compiled with OP_BUILD_TUPLE

```
// Under the hood, (1, "hello", true) compiles to:
// push 1, push "hello", push true, OP_BUILD_TUPLE 3
let metadata = (1, "hello", true)
print(metadata) // Output: (1, "hello", true)
```

7.8 Choosing the Right Collection

Need	Use	Why
Ordered list, may grow	`Array`	Push/pop, index access
Key-value storage	`Map`	O(1) key lookup
Unique elements only	`Set`	Auto-deduplication
Fixed group of values	`Tuple`	Position-based meaning

Table 7.4: Collection type selection guide

Let's see these working together in a realistic scenario:

Listing 7.33: Collections working together

```
// Analyzing survey responses
let responses = [
  ["Alice", "Lattice", 5],
  ["Bob", "Python", 4],
  ["Charlie", "Lattice", 5],
  ["Diana", "Rust", 4],
  ["Eve", "Lattice", 3],
  ["Frank", "Python", 5],
  ["Grace", "Rust", 5]
]

// Group by language
let by_language = responses.group_by(|r| r[1])

// Collect unique languages
let languages = Set::new()
for response in responses {
  languages.add(response[1])
}

// Compute average rating per language
let averages = Map::new()
for lang in languages.to_array() {
  let group = by_language[lang]
  let total = group.reduce(0, |acc, r| acc + r[2])
  averages[lang] = total / group.len()
}

print("Languages surveyed: ${languages}")
print("Average ratings:")
for entry in averages.entries() {
  print("  ${entry[0]}: ${entry[1]}")
}
```

7.9 Exercises

1. **Flatten and Deduplicate.** Given a nested array like `[[1, 2], [2, 3], [3, 4]]`, write a one-liner using `.flatten()` and `.unique()` to produce `[1, 2, 3, 4]`.

2. **Word Frequency Counter.** Write a function that takes a string, splits it into words, and returns a map of word frequencies. Use `group_by` or manual map construction.

3. **Inventory System.** Build an inventory system using a map of item names to quantities. Write functions for `add_stock`, `remove_stock`, and `low_stock_report` (items with fewer than 10 units). Use `.filter()` and `.entries()` in your report function.

4. **Set Operations Challenge.** Given three sets representing students enrolled in Math, Science, and English classes, compute: (a) students taking all three, (b) students taking exactly one class, and (c) students taking Math but not Science.

5. **Data Pipeline.** Given an array of `[name, score]` pairs, write a pipeline using `filter`, `map`, `sort_by`, and `take` to produce a "Top 3 Scorers" leaderboard with formatted strings like `"1. Alice (95)"`.

What's Next?

We've toured Lattice's collection types and their rich method vocabularies. In Chapter 8, we'll zoom in on strings specifically—interpolation, escaping, case transformations, Unicode support, and regular expressions. Strings are Lattice's most method-rich type, and there's a lot to explore.

Chapter 8

Strings in Depth

Strings are the workhorses of most programs. They carry user messages, encode configuration, represent file paths, and compose network protocols. Lattice treats strings as first-class values with a rich set of built-in methods—over two dozen, from basic splitting and trimming to case transformations, character inspection, and regular expression matching.

In this chapter, we'll go beyond the basics of string creation (covered in Chapter 3) and explore the full breadth of what Lattice strings can do.

8.1 Interpolation, Escaping, and Raw Strings

8.1.1 String Interpolation

Lattice strings support interpolation with the `$\{expr\}` syntax. Any expression can appear inside the braces:

Listing 8.1: String interpolation

```
let name = "Lattice"
let version = 1.0

print("Welcome to ${name}!")           // Output: Welcome to Lattice!
print("Version: ${version}")           // Output: Version: 1.0
print("1 + 2 = ${1 + 2}")             // Output: 1 + 2 = 3
print("Name length: ${name.len()}")    // Output: Name length: 7
```

Interpolation works in double-quoted strings and triple-quoted strings. The expression is evaluated, converted to a string, and spliced into the surrounding text. You can nest function calls, method chains, and even conditionals inside the braces:

Listing 8.2: Complex interpolation expressions

```
let scores = [85, 92, 78, 91]
let avg = scores.reduce(0, |a, b| a + b) / scores.len()

print("Average: ${avg} (${if avg >= 90 { "A" } else { "B" }})")
// Output: Average: 86 (B)
```

8.1.2 Escape Sequences

Double-quoted strings support the standard escape sequences:

Escape	Character
`\\n`	Newline
`\\t`	Tab
`\\r`	Carriage return
`\\\\`	Literal backslash
`\\"`	Double quote
`\\'`	Single quote
`\\0`	Null byte
`\\xHH`	Hex byte (two hex digits)

Table 8.1: Escape sequences

Listing 8.3: Using escape sequences

```
print("Line 1\nLine 2")
// Output:
// Line 1
// Line 2

print("Tab\there")
// Output: Tab    here

print("She said \"hello\"")
// Output: She said "hello"

print("Hex: \x41\x42\x43")
// Output: Hex: ABC
```

8.1.3 Single-Quoted Strings

Single-quoted strings work identically to double-quoted strings in Lattice—they support the same escape sequences and interpolation. The choice between `'...'` and `"..."` is purely stylistic.

8.1.4 Triple-Quoted Strings

For multi-line strings, triple quotes preserve whitespace and auto-dedent:

Listing 8.4: Triple-quoted strings

```
let poem = """
    Roses are red,
    Violets are blue,
    Lattice is fast,
    And expressive too.
    """

print(poem)
// Output:
// Roses are red,
// Violets are blue,
// Lattice is fast,
// And expressive too.
```

Triple-quoted strings automatically strip the common leading whitespace from all lines, so you can indent them naturally in your code. They also support interpolation and escape sequences.

8.2 Core String Methods

8.2.1 split — Break a String Apart

The `split` method divides a string by a delimiter:

Listing 8.5: Splitting strings

```
let csv_line = "Alice,30,Engineer"
let fields = csv_line.split(",")
print(fields) // Output: ["Alice", "30", "Engineer"]

let words = "the quick brown fox".split(" ")
print(words) // Output: ["the", "quick", "brown", "fox"]

// Empty delimiter splits into individual characters
let chars = "hello".split("")
print(chars) // Output: ["h", "e", "l", "l", "o"]
```

8.2.2 trim, trim_start, trim_end

These methods remove whitespace from string edges:

Listing 8.6: Trimming whitespace

```
let messy = "   hello, world!   "

print(messy.trim())       // Output: "hello, world!"
print(messy.trim_start()) // Output: "hello, world!   "
print(messy.trim_end())   // Output: "   hello, world!"
```

Whitespace includes spaces, tabs, newlines, and carriage returns.

8.2.3 replace

`replace` substitutes all occurrences of a substring:

Listing 8.7: String replacement

```
let template = "Hello, {name}! Welcome to {place}."

let greeting = template
  .replace("{name}", "Alice")
  .replace("{place}", "Lattice")

print(greeting) // Output: Hello, Alice! Welcome to Lattice.

// Replace all occurrences
let stuttered = "b-b-banana"
print(stuttered.replace("b-", "")) // Output: banana
```

8.2.4 contains, starts_with, ends_with

These methods test whether a string contains, begins with, or ends with a given substring:

Listing 8.8: String searching

```
let filename = "report_2024.pdf"

print(filename.contains("2024"))     // Output: true
print(filename.starts_with("report")) // Output: true
print(filename.ends_with(".pdf"))    // Output: true
print(filename.ends_with(".csv"))    // Output: false
```

8.2.5 count

`count` returns the number of non-overlapping occurrences of a substring:

Listing 8.9: Counting occurrences

```
let text = "abracadabra"

print(text.count("a"))   // Output: 5
print(text.count("abra")) // Output: 2
print(text.count("z"))   // Output: 0
```

8.3 Case Transformations

Lattice provides a complete set of case transformation methods, useful for normalizing identifiers, formatting output, and converting between naming conventions.

Listing 8.10: Basic case conversion

```
let greeting = "Hello, World!"

print(greeting.to_upper()) // Output: HELLO, WORLD!
print(greeting.to_lower()) // Output: hello, world!
```

8.3.1 Naming Convention Converters

These methods convert between common programming naming conventions:

Listing 8.11: Naming convention transformations

```
let identifier = "getUserName"

print(identifier.snake_case())  // Output: get_user_name
print(identifier.kebab_case())  // Output: get-user-name
print(identifier.title_case())  // Output: Get User Name
```

The transformations handle various input formats intelligently:

Listing 8.12: Cross-format conversion

```
// From snake_case
print("user_full_name".camel_case())  // Output: userFullName
print("user_full_name".kebab_case())  // Output: user-full-name
print("user_full_name".title_case())  // Output: User Full Name

// From kebab-case
print("my-component-name".snake_case()) // Output: my_component_name
print("my-component-name".camel_case()) // Output: myComponentName

// capitalize: first letter uppercase, rest lowercase
print("hello world".capitalize()) // Output: Hello world
```

The case detection is aware of camelCase boundaries—when a lowercase letter is followed by an uppercase letter, that's treated as a word boundary. This means `"XMLParser".snake_case()` produces `"x_m_l_parser"` (treating each uppercase letter as a boundary).

Method	Input	Output
`.to_upper()`	`"hello"`	`"HELLO"`
`.to_lower()`	`"HELLO"`	`"hello"`
`.capitalize()`	`"hello world"`	`"Hello world"`
`.snake_case()`	`"getUserName"`	`"get_user_name"`
`.camel_case()`	`"get_user_name"`	`"getUserName"`
`.kebab_case()`	`"getUserName"`	`"get-user-name"`
`.title_case()`	`"hello_world"`	`"Hello World"`

Table 8.2: Case transformation methods

When to Use Case Transformations

Case transformations are especially useful in code generation, template engines, and API adapters. For example, you might convert a database column name like `"first_name"` to `"firstName"` for a JSON API, or to `"First Name"` for a UI label.

8.4 Characters, Bytes, and Substrings

8.4.1 chars() and bytes()

The `chars` method returns an array of individual characters (each as a one-character string), while `bytes` returns an array of byte values:

Listing 8.13: chars and bytes

```
let word = "Hello"

print(word.chars()) // Output: ["H", "e", "l", "l", "o"]
print(word.bytes()) // Output: [72, 101, 108, 108, 111]
```

The `chars` method is useful for character-level processing:

Listing 8.14: Character-level operations

```
fn is_palindrome(s: String) -> Bool {
  let chars = s.to_lower().chars()
  let n = chars.len()
  for i in 0..(n / 2) {
    if chars[i] != chars[n - 1 - i] {
      return false
    }
  }
  true
}

print(is_palindrome("racecar")) // Output: true
print(is_palindrome("hello"))   // Output: false
print(is_palindrome("Madam"))   // Output: true
```

8.4.2 len(), substring(), and index_of()

Listing 8.15: String length, substrings, and searching

```
let message = "Hello, World!"

// Length
print(message.len()) // Output: 13

// Substring extraction (start inclusive, end exclusive)
print(message.substring(0, 5))  // Output: Hello
print(message.substring(7, 12)) // Output: World

// Finding positions
print(message.index_of("World"))  // Output: 7
print(message.index_of("world"))  // Output: -1 (case-sensitive)

// Last occurrence
let repeated = "abcabc"
print(repeated.last_index_of("abc")) // Output: 3
print(repeated.index_of("abc"))      // Output: 0
```

The `substring` method supports negative indices, which count from the end of the string, and automatically clamps out-of-bounds values:

Listing 8.16: Negative indices in substring

```
let text = "Hello, World!"

print(text.substring(-6, -1)) // Output: World
```

8.4.3 More String Utilities

Listing 8.17: reverse, repeat, and padding

```
// Reverse
print("desserts".reverse()) // Output: stressed

// Repeat
print("ha".repeat(3))   // Output: hahaha
print("-".repeat(20))    // Output: --------------------

// Padding
print("42".pad_left(5))       // Output: "   42"
print("42".pad_left(5, "0"))  // Output: "00042"
print("hi".pad_right(10, ".")) // Output: "hi........"
```

Listing 8.18: String validation methods

```
print("hello".is_alpha())        // Output: true
print("hello123".is_alpha())     // Output: false
print("12345".is_digit())        // Output: true
print("abc123".is_alphanumeric()) // Output: true
print("".is_empty())             // Output: true
print("hi".is_empty())           // Output: false
```

8.5 Working with Unicode

Lattice provides two built-in functions for working with character codes: `ord()` and `chr()`.

Method	Returns	Description
`.len()`	`Int`	String length in bytes
`.chars()`	`Array`	Array of characters
`.bytes()`	`Array`	Array of byte values
`.substring(s, e)`	`String`	Extract range `[s, e)`
`.index_of(sub)`	`Int`	First position, or `-1`
`.last_index_of(sub)`	`Int`	Last position, or `-1`
`.reverse()`	`String`	Characters in reverse
`.repeat(n)`	`String`	Repeated `n` times
`.pad_left(w, ch)`	`String`	Left-pad to width
`.pad_right(w, ch)`	`String`	Right-pad to width
`.is_empty()`	`Bool`	Is the string empty?
`.is_alpha()`	`Bool`	All alphabetic?
`.is_digit()`	`Bool`	All digits?
`.is_alphanumeric()`	`Bool`	All alphanumeric?

Table 8.3: Character and substring methods

8.5.1 ord() and chr()

The `ord` function returns the numeric code of a character, and `chr` returns the character for a given code:

Listing 8.19: Converting between characters and codes

```
print(ord("A"))  // Output: 65
print(ord("a"))  // Output: 97
print(ord("0"))  // Output: 48
print(ord("\n")) // Output: 10

print(chr(65))  // Output: A
print(chr(97))  // Output: a
print(chr(48))  // Output: 0
```

ord() and chr()

`ord(s)` returns the code point of the first character in string `s`. If the string is empty, it returns `-1`. `chr(n)` returns a one-character string for the code point `n` (currently limited to 0–127, the ASCII range).

These functions are useful for character classification, encoding, and building custom parsers:

Listing 8.20: Character classification with ord

```
fn is_uppercase(ch: String) -> Bool {
  let code = ord(ch)
  code >= 65 && code <= 90
}

fn is_lowercase(ch: String) -> Bool {
  let code = ord(ch)
  code >= 97 && code <= 122
}

fn rot13(text: String) -> String {
  let chars = text.chars()
  flux result = ""
  for ch in chars {
    let code = ord(ch)
    if is_uppercase(ch) {
      result = result + chr((code - 65 + 13) % 26 + 65)
    } else if is_lowercase(ch) {
      result = result + chr((code - 97 + 13) % 26 + 97)
    } else {
      result = result + ch
    }
  }
  result
}

print(rot13("Hello, World!")) // Output: Uryyb, Jbeyq!
print(rot13("Uryyb, Jbeyq!")) // Output: Hello, World!
```

8.5.2 Building a Caesar Cipher

Listing 8.21: A simple Caesar cipher

```
fn caesar_encrypt(text: String, shift: Int) -> String {
  flux encrypted = ""
  for ch in text.chars() {
    let code = ord(ch)
    if code >= 65 && code <= 90 {
      // Uppercase letter
      encrypted = encrypted + chr((code - 65 + shift) % 26 + 65)
    } else if code >= 97 && code <= 122 {
      // Lowercase letter
      encrypted = encrypted + chr((code - 97 + shift) % 26 + 97)
    } else {
      encrypted = encrypted + ch
    }
  }
  encrypted
}

fn caesar_decrypt(text: String, shift: Int) -> String {
  caesar_encrypt(text, 26 - shift)
}

let secret = caesar_encrypt("Attack at dawn", 3)
print(secret)                       // Output: Dwwdfn dw gdzq
print(caesar_decrypt(secret, 3))    // Output: Attack at dawn
```

8.6 Regular Expressions

Lattice includes built-in regular expression support via three string methods: `regex_match`, `regex_find_all`, and `regex_replace`. These use POSIX extended regular expressions under the hood (implemented in `src/regex_ops.c`).

8.6.1 regex_match — Test a Pattern

`regex_match` tests whether a string matches a regular expression pattern:

Listing 8.22: Pattern matching with regex

```
let email = "alice@example.com"

print(email.regex_match("[a-zA-Z]+@[a-zA-Z]+\\.[a-zA-Z]+"))
// Output: true

print("not-an-email".regex_match("[a-zA-Z]+@[a-zA-Z]+\\.[a-zA-Z]+"))
// Output: false

// Validate a phone number format
let phone = "555-123-4567"
print(phone.regex_match("[0-9]{3}-[0-9]{3}-[0-9]{4}"))
// Output: true
```

8.6.2 regex_find_all — Extract All Matches

`regex_find_all` returns an array of all substrings matching the pattern:

Listing 8.23: Finding all regex matches

```
let text = "Call 555-1234 or 555-5678 for info"

let phone_numbers = text.regex_find_all("[0-9]{3}-[0-9]{4}")
print(phone_numbers) // Output: ["555-1234", "555-5678"]

// Extract all words
let words = "Hello, World! How are you?".regex_find_all("[a-zA-Z]+")
print(words) // Output: ["Hello", "World", "How", "are", "you"]

// Find all numbers (including decimals)
let data = "Price: $12.50, Tax: $1.00, Total: $13.50"
let amounts = data.regex_find_all("[0-9]+\\.[0-9]+")
print(amounts) // Output: ["12.50", "1.00", "13.50"]
```

8.6.3 regex_replace — Pattern-Based Replacement

`regex_replace` replaces all matches of a pattern with a replacement string:

Listing 8.24: Regex replacement

```
// Normalize whitespace
let messy = "too   many    spaces   here"
let clean = messy.regex_replace("[ ]+", " ")
print(clean) // Output: too many spaces here

// Remove all non-alphanumeric characters
let raw = "Hello, World! (2024)"
let cleaned = raw.regex_replace("[^a-zA-Z0-9 ]", "")
print(cleaned) // Output: Hello World 2024

// Redact phone numbers
let log = "User called from 555-1234, callback to 555-5678"
let redacted = log.regex_replace("[0-9]{3}-[0-9]{4}", "XXX-XXXX")
print(redacted) // Output: User called from XXX-XXXX, callback to XXX-XXXX
```

8.6.4 Regex Flags

All three regex methods accept an optional flags string as the last argument:

Listing 8.25: Case-insensitive regex

```
let text = "Hello HELLO hello HeLLo"

// Case-insensitive matching
let matches = text.regex_find_all("hello", "i")
print(matches) // Output: ["Hello", "HELLO", "hello", "HeLLo"]

// Case-insensitive replacement
let normalized = text.regex_replace("hello", "hi", "i")
print(normalized) // Output: hi hi hi hi
```

The available flags are:

- `"i"`: Case-insensitive matching
- `"m"`: Multiline mode (newlines treated as line boundaries)

POSIX Regex Syntax

Lattice uses POSIX extended regular expressions, not Perl-compatible (PCRE) syntax. This means some patterns you may be used to from other languages work differently:

- Use `[0-9]` instead of `\\d`
- Use `[a-zA-Z0-9_]` instead of `\\w`
- Lookaheads and lookbehinds are not supported

8.7 Putting It All Together

Let's combine these string methods into a practical example—a log file analyzer:

Listing 8.26: A log file analyzer

```
fn analyze_log(log_text: String) -> Map {
  let lines = log_text.split("\n").filter(|l| !l.is_empty())
  let results = Map::new()

  // Count log levels
  let levels = Map::new()
  levels["INFO"] = 0
  levels["WARN"] = 0
  levels["ERROR"] = 0

  for line in lines {
    if line.contains("INFO") {
      levels["INFO"] = levels["INFO"] + 1
    } else if line.contains("WARN") {
      levels["WARN"] = levels["WARN"] + 1
    } else if line.contains("ERROR") {
      levels["ERROR"] = levels["ERROR"] + 1
    }
  }
  results["levels"] = levels
  results["total_lines"] = lines.len()

  // Extract timestamps
  let timestamps = []
  for line in lines {
    let ts_matches = line.regex_find_all("[0-9]{4}-[0-9]{2}-[0-9]{2}")
    for ts in ts_matches {
      timestamps.push(ts)
    }
  }
  results["dates"] = timestamps.unique()

  // Extract error messages
  let errors = lines
    .filter(|l| l.contains("ERROR"))
    .map(|l| {
      let parts = l.split("ERROR")
      if parts.len() > 1 {
        parts[1].trim()
      } else {
        l
      }
    })
  results["errors"] = errors

  results
}
```

8.8 Exercises

1. **String Compression.** Write a function that performs basic run-length encoding: `"aaabbbcc"` becomes `"a3b3c2"`. Handle single characters (no count needed for runs of 1).
2. **CSV Parser.** Write a function that takes a CSV string (with header row) and returns an array of maps, where each map represents a row with column names as keys. Handle quoted fields if possible.
3. **Slug Generator.** Write a function that converts a title like `"Hello, World! This is Great"` into a URL slug like `"hello-world-this-is-great"`. Use `to_lower`, `regex_replace`, and `trim`.
4. **Password Validator.** Write a function that checks if a password meets these criteria: at least 8 characters, contains at least one uppercase letter, one lowercase letter, one digit, and one special character. Use `regex_match` or character inspection.
5. **Template Engine.** Write a simple template engine where `render("Hello \{\{name\}\}, age \{\{age\}\}", context)` replaces `\{\{name\}\}` and `\{\{age\}\}` with values from a context map. Use `regex_find_all` to find placeholders and `replace` to fill them in.

What's Next?

Strings give us the ability to represent and manipulate text, but programs often need to inspect the *structure* of data—not just its content. In Chapter 9, we'll explore Lattice's `match` expression, which lets you destructure arrays, structs, and enums, test ranges, and make decisions based on the shape of your data. Pattern matching is one of Lattice's most powerful features, and it builds naturally on everything you've learned so far.

Chapter 9

Pattern Matching

Every program makes decisions. In Chapter 5 we used `if`/`else` chains to steer execution based on boolean conditions, and that works beautifully when you have one or two tests to run. But programs often face a harder question: "What *shape* does this data have?" Is the HTTP response a success or a failure? Does the command-line argument look like a number, a flag, or a filename? Is the list empty, a singleton, or something longer?

Pattern matching answers these questions directly. Instead of writing a cascade of `if`/`else` branches—each one manually testing a field or comparing a value—you describe what the data *looks like*, and Lattice picks the first description that fits.

Listing 9.1: A first taste of match

```
let status_code = 404

let message = match status_code {
    200 => "OK",
    301 => "Moved Permanently",
    404 => "Not Found",
    500 => "Internal Server Error",
    _   => "Unknown status"
}

print(message) // Not Found
```

This chapter builds from that starting point. We will explore every pattern type Lattice supports—from literals and wildcards through range patterns, array destructuring, struct field extraction, and

enum variant matching. Along the way we will meet *guards*, *bindings*, and *exhaustiveness checking*, the compiler's way of telling you when you have forgotten a case.

9.1 The `match` Expression

A `match` expression evaluates a *scrutinee* (the value being inspected) and compares it against a series of *arms*. Each arm has a pattern on the left side of `=>` and a body on the right:

Match Expression

```
match scrutinee {
    pattern1 => body1,
    pattern2 => body2,
    _        => fallback_body
}
```

A *match expression* evaluates the scrutinee once, then tries each arm's pattern from top to bottom. The body of the first matching arm is executed, and its value becomes the value of the entire `match` expression. If no arm matches, the expression evaluates to `nil`.

Because `match` is an expression, you can bind its result to a variable, return it from a function, or embed it anywhere a value is expected:

Listing 9.2: match as an expression

```
fn describe_temperature(temp: Int) -> String {
    return match temp {
        0   => "Freezing point of water",
        100 => "Boiling point of water",
        _   => "Just another temperature: ${temp}"
    }
}

print(describe_temperature(100)) // Boiling point of water
print(describe_temperature(42))  // Just another temperature: 42
```

9.1.1 Arm Bodies: Expressions and Blocks

An arm body can be a single expression (as we have seen) or a block enclosed in curly braces. When a block is used, the value of the last expression in the block becomes the arm's value:

Listing 9.3: Block bodies in match arms

```
let score = 87

let grade = match score {
    100 => "Perfect!",
    _ => {
        let letter = if score >= 90 { "A" } else { "B" }
        "${letter} (${score}/100)"
    }
}

print(grade) // B (87/100)
```

Commas Between Arms

Arms are separated by commas. The trailing comma after the last arm is optional but encouraged for consistency—it makes adding new arms later easier and produces cleaner diffs in version control.

9.1.2 First Match Wins

Arms are tried from top to bottom, and the first one that matches wins. This ordering matters when patterns overlap:

Listing 9.4: Order matters—first match wins

```
let n = 0

let result = match n {
    0 => "zero",       // This arm matches first
    _ => "something",  // This arm would also match, but never reached
}

print(result) // zero
```

If you put a wildcard _ before more specific patterns, the specific patterns will never be reached. Lattice currently does not warn about unreachable arms, so you should arrange your patterns from most specific to least specific.

No Match Returns nil

If no arm matches the scrutinee, the entire `match` expression evaluates to `nil`—not an error. This is a deliberate design choice that keeps programs running, but it also means a missing arm might silently produce `nil` where you expected a real value. The exhaustiveness checker (Section 9.9) helps catch these situations at compile time.

9.2 Literal Patterns

The most direct kind of pattern matches a specific, concrete value. Literal patterns support integers, floats, strings, booleans, and `nil`:

Listing 9.5: Literal patterns across types

```
fn classify(value: any) -> String {
    return match value {
        42      => "The answer",
        3.14    => "Approximately pi",
        "hello" => "A greeting",
        true    => "Affirmative",
        nil     => "Nothing at all",
        _       => "Something else"
    }
}

print(classify(42))      // The answer
print(classify("hello")) // A greeting
print(classify(nil))     // Nothing at all
print(classify(99))      // Something else
```

Literal patterns use value equality to compare. Two integers are equal if they hold the same number; two strings are equal if they contain the same characters. Complex types like arrays and structs are *not* compared by value in literal patterns—use destructuring patterns (Sections 9.6 and 9.7) for those.

Negative numeric literals work as you would expect:

Listing 9.6: Negative literal patterns

```
let temp = -40

let note = match temp {
    -40 => "Where Celsius meets Fahrenheit",
    0   => "Freezing",
    _   => "Unremarkable"
}

print(note) // Where Celsius meets Fahrenheit
```

9.3 Wildcard and Binding Patterns

9.3.1 The Wildcard _

The underscore _ matches any value without capturing it. It is the catch-all, the "I don't care what this is" pattern:

Listing 9.7: Wildcard as a catch-all

```
fn is_special(n: Int) -> Bool {
    return match n {
        0 => true,
        1 => true,
        _ => false
    }
}
```

A wildcard arm placed at the end of a `match` guarantees that every possible value is covered, which makes the match exhaustive.

9.3.2 Binding Patterns

Sometimes you want to match *any* value, but you also want to use it inside the arm's body. A binding pattern does exactly that—any bare identifier (other than _) becomes a variable bound to the scrutinee's value:

Listing 9.8: Binding patterns capture the scrutinee

```
let response_code = 418

let msg = match response_code {
    200 => "Success",
    404 => "Not found",
    code => "Unexpected status: ${code}"
}

print(msg) // Unexpected status: 418
```

The name `code` in that last arm is not a keyword or a previously defined variable—it is a fresh local variable whose value is the scrutinee (here, `418`). The binding is only available inside the arm's body.

Bindings vs. Literals

How does Lattice tell the difference between a binding pattern like `code` and a literal pattern? The rule is: literal patterns are actual literal values (`42`, `"hello"`, `true`, `nil`). Any other bare identifier is treated as a binding. You cannot match against the value of an existing variable by naming it in a pattern—use a guard instead (Section 9.5).

Binding patterns truly shine when combined with guards, which we will explore in Section 9.5.

9.4 Range Patterns

Range patterns match values that fall within a numeric range. The syntax uses the familiar `..` operator, but with an important twist: range patterns are **inclusive** on both ends.

Listing 9.9: Grading with range patterns

```
fn letter_grade(score: Int) -> String {
    return match score {
        90..100 => "A",
        80..89  => "B",
        70..79  => "C",
        60..69  => "D",
        0..59   => "F",
        _       => "Invalid score"
    }
}

print(letter_grade(95))  // A
print(letter_grade(80))  // B
print(letter_grade(70))  // C
print(letter_grade(42))  // F
```

Ranges in Patterns vs. Ranges Elsewhere

The `..` operator has different semantics depending on context. When used to create a `Range` value (for example, in a `for` loop), `a..b` is *half-open*: it includes `a` but excludes `b`. Inside a `match` pattern, however, `a..b` is *inclusive*: it matches values where $a \leq \text{value} \leq b$. This is a deliberate choice—inclusive ranges make match arms more intuitive when partitioning a numeric domain into adjacent regions.

Under the hood, the compiler translates a range pattern into a pair of comparisons. The pattern `80..89` becomes roughly `value >= 80 && value <= 89`, using `OP_GTEQ` and `OP_LTEQ` opcodes connected by a short-circuit jump. You can see this in the range-pattern compilation logic in `src/stackcompiler.c`.

Range patterns work with integer values. You can also use variables or expressions as the endpoints:

Listing 9.10: Dynamic range endpoints

```
let threshold = 50

let category = match 42 {
    0..threshold => "Below or at threshold",
    _            => "Above threshold"
}

print(category) // Below or at threshold
```

9.5 Guards

A *guard* is an additional boolean condition attached to a match arm with the `if` keyword. The arm only matches if both the pattern and the guard are satisfied:

Listing 9.11: Guards refine pattern matches

```
fn classify_number(n: Int) -> String {
    return match n {
        0 => "zero",
        x if x > 0 => "positive: ${x}",
        x if x < 0 => "negative: ${x}",
        _ => "unreachable"
    }
}

print(classify_number(42))  // positive: 42
print(classify_number(-7))  // negative: -7
print(classify_number(0))   // zero
```

In the arms `x if x > 0` and `x if x < 0`, the identifier `x` is a binding pattern that captures the scrutinee, and the `if` clause is a guard that tests the captured value. The guard expression can use any bound variables from the pattern, making it possible to write powerful, flexible conditions.

Guards are full expressions, so you can call functions, combine boolean operators, or access methods:

Listing 9.12: Complex guard expressions

```
fn describe_name(name: String) -> String {
    return match name {
        s if s.len() == 0    => "Empty name",
        s if s.len() > 50    => "That's a very long name",
        s if s.contains(" ") => "Full name: ${s}",
        s                    => "First name only: ${s}"
    }
}

print(describe_name(""))                // Empty name
print(describe_name("Ada"))             // First name only: Ada
print(describe_name("Ada Lovelace"))    // Full name: Ada Lovelace
```

Guards and Exhaustiveness

A guarded arm is not considered a catch-all by the exhaustiveness checker, even if its pattern is a wildcard or a binding. The guard might evaluate to `false`, so the checker cannot guarantee the arm will match. To ensure full coverage, add an unguarded catch-all arm at the end.

9.5.1 Guards with Other Pattern Types

Guards can be combined with any pattern type. Here is a range pattern with a guard:

Listing 9.13: Combining range patterns and guards

```
fn shipping_message(weight: Int) -> String {
    return match weight {
        1..5 if weight == 1   => "Letter rate",
        1..5                  => "Small parcel",
        6..30                 => "Standard parcel",
        _ if weight > 1000    => "Freight only",
        _                     => "Large parcel"
    }
}

print(shipping_message(1))    // Letter rate
print(shipping_message(3))    // Small parcel
print(shipping_message(2000)) // Freight only
```

9.6 Array Destructuring Patterns

So far, every pattern has matched a single flat value. Array patterns reach *inside* a value, matching the structure and extracting individual elements:

Listing 9.14: Basic array destructuring

```
let point = [10, 20]

match point {
    [0, 0]    => print("Origin"),
    [x, 0]    => print("On the x-axis at ${x}"),
    [0, y]    => print("On the y-axis at ${y}"),
    [x, y]    => print("Point at (${x}, ${y})")
}
// Point at (10, 20)
```

An array pattern is enclosed in square brackets and contains sub-patterns for each element. The scrutinee must be an array, and—without a rest pattern—its length must exactly match the number of elements in the pattern.

Each element position can hold any sub-pattern:

- A **literal** like `0` matches that specific value at that position.
- A **binding** like `x` matches any value and captures it.
- A **wildcard** `_` matches any value without capturing.

Listing 9.15: Mixing literal and binding sub-patterns

```
let rgb = [255, 128, 0]

let color_name = match rgb {
    [255, 0, 0]     => "Red",
    [0, 255, 0]     => "Green",
    [0, 0, 255]     => "Blue",
    [255, 255, 255] => "White",
    [0, 0, 0]       => "Black",
    [r, g, b]       => "Custom RGB(${r}, ${g}, ${b})"
}

print(color_name) // Custom RGB(255, 128, 0)
```

9.6.1 Rest Patterns

What if you do not know (or care about) how many elements the array has? The rest pattern `...name` collects zero or more remaining elements into a sub-array:

Listing 9.16: Rest patterns collect remaining elements

```
let numbers = [1, 2, 3, 4, 5]

match numbers {
    []           => print("Empty"),
    [only]       => print("Just one: ${only}"),
    [first, ...rest] => {
        print("First: ${first}")
        print("Rest: ${rest}")
    }
}
// First: 1
// Rest: [2, 3, 4, 5]
```

The rest pattern is powerful for processing lists head-first. You can place elements *after* the rest pattern too, matching from both ends:

Listing 9.17: Rest with trailing elements

```
let log_entries = ["INFO", "Starting up", "v2.1", "2024-01-15"]

match log_entries {
    [level, ...middle, date] => {
        print("Level: ${level}")
        print("Date: ${date}")
        print("Middle: ${middle}")
    },
    _ => print("Unexpected format")
}
// Level: INFO
// Date: 2024-01-15
// Middle: [Starting up, v2.1]
```

The compiler handles post-rest elements by indexing from the end of the array. For a pattern like `[first, ..mid, last]`, the element `last` is extracted at index `array.len() - 1`, and `mid` receives a slice of everything in between.

Length Requirements with Rest

Without a rest pattern, the array length must match the pattern count exactly. With a rest pattern, the array must have *at least* as many elements as the non-rest positions. A pattern `[a, ..rest, b]` requires at least two elements; the rest can be empty.

9.6.2 Array Patterns and Guards

Array patterns combine naturally with guards for precise conditions:

Listing 9.18: Array patterns with guards

```
fn describe_pair(pair: Array) -> String {
    return match pair {
        [a, b] if a == b     => "Equal pair: ${a}",
        [a, b] if a + b == 0 => "Opposite pair: ${a} and ${b}",
        [a, b]               => "Pair: ${a} and ${b}",
        _                    => "Not a pair"
    }
}

print(describe_pair([5, 5]))    // Equal pair: 5
print(describe_pair([3, -3]))   // Opposite pair: 3 and -3
print(describe_pair([1, 2]))    // Pair: 1 and 2
```

9.7 Struct Destructuring Patterns

Struct patterns match values by their field names and optionally by field values. They use curly braces and are especially useful for pulling out the fields you need without cluttering your code with manual field access.

Listing 9.19: Basic struct destructuring

```
struct User {
    name: String,
    age: Int,
    active: Bool
}

let user = User { name: "Rosalind", age: 33, active: true }

match user {
    {active: false} => print("Inactive user"),
    {name, age}     => print("${name}, age ${age}")
}
// Rosalind, age 33
```

There are two forms of struct field patterns:

1. **Shorthand binding**: Writing just a field name like `{name, age}` matches any value for those fields and binds them to variables of the same name.

2. **Value matching**: Writing `{active: false}` matches only when the `active` field equals `false`.

You can mix both forms in a single pattern:

Listing 9.20: Mixed shorthand and value patterns

```
struct Config {
    mode: String,
    debug: Bool,
    port: Int
}

let config = Config { mode: "production", debug: false, port: 8080 }

match config {
    {mode: "test", debug: true} => print("Test mode with debugging"),
    {mode: "production", port}  => print("Production on port ${port}"),
    {mode}                      => print("Running in ${mode} mode")
}
// Production on port 8080
```

Partial Matching

Struct patterns do not need to mention every field. A pattern like `\{name\}` matches any struct that has a `name` field, regardless of what other fields it contains or what their values are. This makes struct patterns resilient to additions—if you add a field to the struct later, existing patterns continue to work.

9.8 Enum Variant Patterns

Enum variant patterns are where pattern matching truly shines. They let you distinguish between an enum's variants and extract the data each variant carries:

Listing 9.21: Matching enum variants

```
enum Shape {
    Circle(Float),
    Rectangle(Float, Float),
    Triangle(Float, Float, Float)
}

fn area(shape: Shape) -> Float {
    return match shape {
        Shape::Circle(r)           => 3.14159 * r * r,
        Shape::Rectangle(w, h)     => w * h,
        Shape::Triangle(a, b, c)   => {
            let s = (a + b + c) / 2.0
            // Heron's formula
            let val = s * (s - a) * (s - b) * (s - c)
            // Approximate square root via Newton's method
            flux guess = val / 2.0
            for _ in 0..10 {
                guess = (guess + val / guess) / 2.0
            }
            guess
        }
    }
}

print(area(Shape::Circle(5.0)))          // 78.53975
print(area(Shape::Rectangle(3.0, 4.0)))  // 12.0
```

The pattern `Shape::Circle(r)` checks three things: (1) the value is an enum, (2) it belongs to the `Shape` enum, and (3) its variant is `Circle`. If all three pass, the payload is bound to `r`.

9.8.1 Variants Without Payloads

For unit variants (those carrying no data), use the variant name without parentheses:

Listing 9.22: Matching unit variants

```
enum Direction {
    North,
    South,
    East,
    West
}

fn arrow(dir: Direction) -> String {
    return match dir {
        Direction::North => "^",
        Direction::South => "v",
        Direction::East  => ">",
        Direction::West  => "<"
    }
}

print(arrow(Direction::North)) // ^
print(arrow(Direction::West))  // <
```

9.8.2 Nested Enum Patterns

Each payload position in an enum variant pattern can itself be a sub-pattern—a literal to match a specific value, a binding to capture the value, or a wildcard to ignore it:

Listing 9.23: Sub-patterns in enum payloads

```
enum Result {
    Ok(any),
    Err(String)
}

fn handle(result: Result) -> String {
    return match result {
        Result::Ok(0)      => "Zero result",
        Result::Ok(value)  => "Got: ${value}",
        Result::Err(msg)   => "Error: ${msg}"
    }
}

print(handle(Result::Ok(0)))            // Zero result
print(handle(Result::Ok(42)))           // Got: 42
print(handle(Result::Err("timeout")))   // Error: timeout
```

9.8.3 Enum Matching with Guards

Guards work seamlessly with enum patterns:

Listing 9.24: Guards on enum patterns

```
enum Temperature {
    Celsius(Float),
    Fahrenheit(Float)
}

fn describe(temp: Temperature) -> String {
    return match temp {
        Temperature::Celsius(c) if c > 100.0    => "Boiling! (${c}C)",
        Temperature::Celsius(c) if c < 0.0      => "Freezing! (${c}C)",
        Temperature::Celsius(c)                  => "Moderate (${c}C)",
        Temperature::Fahrenheit(f) if f > 212.0  => "Boiling! (${f}F)",
        Temperature::Fahrenheit(f)               => "Tolerable (${f}F)"
    }
}

print(describe(Temperature::Celsius(105.0)))    // Boiling! (105.0C)
print(describe(Temperature::Fahrenheit(72.0)))  // Tolerable (72.0F)
```

9.9 Exhaustiveness Checking

A match expression is *exhaustive* if it handles every possible value the scrutinee might have. Lattice's compiler includes a static analysis pass (implemented in `src/match_check.c`) that inspects every `match` expression in your program and warns you when coverage is incomplete.

9.9.1 How the Checker Works

The exhaustiveness checker runs after parsing, before code generation. It walks the entire AST—including nested functions, `impl` blocks, and `test` blocks—looking for `match` expressions. For each one, it asks a series of questions:

1. **Are there zero arms?** If so, the checker emits: `warning: match expression on line N has no arms`.

2. **Is there a catch-all arm?** An unguarded wildcard (`_`) or an unguarded binding pattern (`x`) without a phase qualifier covers every remaining case. If one exists, the match is exhaustive.

3. **Are we matching an enum?** If any arm uses an enum variant pattern, the checker looks up the enum declaration, builds a coverage set, and reports missing variants.

4. **Are we matching booleans?** If all patterns are boolean literals, the checker verifies both `true` and `false` are present.

5. **Everything else**: For integer, float, or string scrutinees without a catch-all, the checker recommends: `consider adding a wildcard _ arm`.

9.9.2 Enum Exhaustiveness

Enum exhaustiveness is where the checker is most precise. Consider a `Color` enum:

Listing 9.25: The checker catches missing variants

```
enum Color {
    Red,
    Green,
    Blue
}

let c = Color::Red

// This produces a warning:
// warning: non-exhaustive match: missing Color variant `Color::Blue`
let name = match c {
    Color::Red   => "red",
    Color::Green => "green"
}
```

The warning names the exact variant you forgot. If multiple variants are missing, they are all listed:

```
// warning: non-exhaustive match: missing Color variants
//          `Color::Green`, `Color::Blue`
```

To satisfy the checker, either handle every variant or add a wildcard arm:

Listing 9.26: Making an enum match exhaustive

```
let name = match c {
    Color::Red   => "red",
    Color::Green => "green",
    Color::Blue  => "blue"
}
// No warning --- all variants covered
```

9.9.3 Boolean Exhaustiveness

When every arm uses a boolean literal, the checker ensures both `true` and `false` are covered:

Listing 9.27: Boolean exhaustiveness

```
let flag = true

// warning: non-exhaustive match: missing `false` case
let label = match flag {
    true => "on"
}

// Fixed:
let label = match flag {
    true  => "on",
    false => "off"
}
```

9.9.4 Limitations of the Checker

The exhaustiveness checker is deliberately pragmatic rather than exhaustive itself. A few limitations to be aware of:

- **No redundancy warnings**: The checker does not report unreachable or redundant arms. If you put a wildcard before specific patterns, no warning is issued.
- **No deep analysis of arrays or structs**: Array and struct patterns are not analyzed for completeness. The checker relies on a wildcard arm for these cases.
- **Guarded arms and enum coverage**: A guarded enum arm is still counted toward variant coverage. The checker pragmatically assumes that if you wrote a pattern for a variant with a guard, you have considered that variant.
- **Range patterns are not analyzed**: The checker does not verify that a set of range patterns covers the entire integer domain.

A Good Habit

Even when the checker does not require it, adding a wildcard arm with a meaningful response (or an error) is good defensive programming. Future changes to an enum or data format are less likely to introduce silent bugs if there is a catch-all that handles the unexpected.

9.10 Phase-Qualified Patterns

Lattice's phase system (explored in depth in Chapter 11) assigns every value a *phase*—fluid, crystal, or unphased. Pattern matching can discriminate on phase by prefixing a pattern with a phase qualifier:

Listing 9.28: Phase-qualified patterns

```
flux temperature = 72
fix boiling_point = 212

let status = match temperature {
    crystal _ => "This value is crystallized",
    fluid t   => "Fluid value: ${t}"
}

print(status) // Fluid value: 72

let bp_status = match boiling_point {
    crystal t => "Crystal: ${t}",
    fluid _   => "Fluid"
}

print(bp_status) // Crystal: 212
```

The `fluid` qualifier matches values in the fluid or unphased state (which is the default). The `crystal` qualifier matches only values that have been crystallized with `fix` or `freeze`.

Phase Qualifiers and Exhaustiveness

A phase-qualified wildcard or binding is *not* considered a catch-all by the exhaustiveness checker. The pattern `crystal _` only matches crystal values, so it cannot guarantee coverage of all inputs. You still need an unqualified catch-all arm.

Phase-qualified patterns are a niche feature. They become useful in generic code that must behave differently depending on whether a value is mutable or immutable, which we will explore further in Chapter 11.

9.11 Putting It All Together

Let us build a more realistic example that combines multiple pattern types. Imagine we are writing a command-line argument parser:

Listing 9.29: A command-line argument processor

```
fn process_args(args: Array) -> String {
    return match args {
        []                    => "No arguments provided. Use --help.",
        ["--help"]            => "Usage: app [options] [files...]",
        ["--version"]         => "v2.1.0",
        ["--output", path]    => "Output will be written to ${path}",
        ["--verbose", ...rest] => {
            let files = rest.join(", ")
            "Verbose mode, processing: ${files}"
        },
        [cmd, ...rest] => "Unknown command: ${cmd}"
    }
}

print(process_args([]))
// No arguments provided. Use --help.

print(process_args(["--help"]))
// Usage: app [options] [files...]

print(process_args(["--output", "/tmp/out.txt"]))
// Output will be written to /tmp/out.txt

print(process_args(["--verbose", "a.txt", "b.txt"]))
// Verbose mode, processing: a.txt, b.txt
```

Here is another example that processes a stream of events using enum variants:

Listing 9.30: Processing events with enum matching

```
enum Event {
    Click(Int, Int),
    KeyPress(String),
    Resize(Int, Int),
    Close
}

fn handle_event(event: Event) -> String {
    return match event {
        Event::Click(x, y) if x < 0 => "Click out of bounds",
        Event::Click(0, 0)          => "Click at origin",
        Event::Click(x, y)          => "Click at (${x}, ${y})",
        Event::KeyPress("q")        => "Quit requested",
        Event::KeyPress("h")        => "Help requested",
        Event::KeyPress(key)        => "Key: ${key}",
        Event::Resize(w, h)         => "Window is now ${w}x${h}",
        Event::Close                => "Goodbye!"
    }
}

print(handle_event(Event::Click(100, 200)))
// Click at (100, 200)

print(handle_event(Event::KeyPress("q")))
// Quit requested

print(handle_event(Event::Close))
// Goodbye!
```

And finally, an example that matches struct data with guards:

Listing 9.31: Struct matching for access control

```
struct Request {
    method: String,
    path: String,
    authenticated: Bool
}

fn route(req: Request) -> String {
    return match req {
        {method: "GET", path: "/"}          => "Home page",
        {method: "GET", path: "/health"}    => "OK",
        {method: "GET", path}               => "Serving ${path}",
        {method: "POST", authenticated: false} => "401 Unauthorized",
        {method: "POST", path}              => "Processing POST to ${path}",
        {method}                            => "405 Method Not Allowed: ${method}"
    }
}

let req = Request { method: "POST", path: "/api/data", authenticated: true }
print(route(req)) // Processing POST to /api/data

let bad_req = Request { method: "POST", path: "/api/data", authenticated: false }
print(route(bad_req)) // 401 Unauthorized
```

9.12 How Match Compiles

Understanding how `match` compiles helps you reason about performance and appreciate the elegance of the design. Lattice has **no dedicated match opcodes** in its bytecode instruction set. Instead, the compiler in `src/stackcompiler.c` translates each match expression into a sequence of general-purpose opcodes—comparisons, jumps, and stack manipulations.

9.12.1 The Stack Invariant

The compiler maintains a key invariant: the scrutinee stays on the stack across all arms. Before testing each arm, the compiler emits an `OP_DUP` to duplicate the scrutinee, then compares the duplicate against the pattern. If the arm does not match, the duplicate is popped and we move on. If it does match, the arm body executes, and the scrutinee is cleaned up afterward.

9.12.2 Compilation by Pattern Type

Literal patterns compile to a `OP_DUP` followed by pushing the literal constant and an `OP_EQ` comparison.

Wildcard patterns without a phase qualifier push `OP_TRUE` directly—they always match.

Range patterns compile to two comparisons (`OP_GTEQ` for the start, `OP_LTEQ` for the end) connected by a short-circuit jump.

Binding patterns duplicate the scrutinee and create a local variable slot. Guard compilation emits the guard expression, and the result is tested with `OP_JUMP_IF_FALSE`.

Complex patterns (arrays, structs, enums) use a two-phase approach:

1. **Structural check**: Verify the type (using the `typeof` builtin), check length or field names, and compare literal sub-patterns. Multiple checks are chained with short-circuit `OP_JUMP_IF_FALSE` jumps, so if the first check fails, no further work is done.

2. **Binding extraction**: If the structural check passes, extract the needed elements. Array elements are pulled out with `OP_INDEX`, struct fields with `OP_GET_FIELD`, and enum payloads by calling the `.payload()` method followed by `OP_INDEX`.

After the matched arm's body executes, an `OP_JUMP` skips over all remaining arms. At the very end, a fallback emits `OP_POP` (to discard the scrutinee) and `OP_NIL` for the no-match case.

Performance Implications

Because each arm is compiled as a linear sequence of checks and jumps, match expressions are evaluated in $O(n)$ time where n is the number of arms. The compiler does not build a jump table or decision tree. For small match expressions (the common case), this is perfectly efficient. For matches with many literal arms, the short-circuit jumps keep things fast by skipping unnecessary work.

9.13 Common Patterns and Idioms

9.13.1 Replacing Long if/else Chains

Any time you find yourself writing three or more `if`/`else` branches that all test the same variable, consider using `match` instead:

Listing 9.32: Before and after: if/else to match

```
// Before
fn day_type(day: String) -> String {
    if day == "Saturday" {
        return "weekend"
    } else if day == "Sunday" {
        return "weekend"
    } else {
        return "weekday"
    }
}

// After
fn day_type(day: String) -> String {
    return match day {
        "Saturday" => "weekend",
        "Sunday"   => "weekend",
        _          => "weekday"
    }
}
```

The `match` version is more compact, makes the structure of the decision visible at a glance, and ensures every case is handled.

9.13.2 Extracting from Nested Structures

Match excels at pulling values out of nested data:

Listing 9.33: Extracting nested data

```
enum Response {
    Success(Array),
    Error(String)
}

fn first_item(resp: Response) -> String {
    return match resp {
        Response::Success([first, ...rest]) => "First: ${first}",
        Response::Success([])               => "Empty results",
        Response::Error(msg)                => "Failed: ${msg}",
        _                                   => "Unknown"
    }
}

print(first_item(Response::Success([10, 20, 30])))
// First: 10

print(first_item(Response::Success([])))
// Empty results

print(first_item(Response::Error("timeout")))
// Failed: timeout
```

9.13.3 State Machines

Enums and match are a natural way to model state machines. Each state is a variant, and each transition is a match arm:

Listing 9.34: A state machine with match

```
enum ConnectionState {
    Disconnected,
    Connecting(String),
    Connected(String),
    Error(String)
}

fn next_state(state: ConnectionState, action: String) -> ConnectionState {
    return match state {
        ConnectionState::Disconnected if action == "connect" => {
            ConnectionState::Connecting("server.example.com")
        },
        ConnectionState::Connecting(host) if action == "success" => {
            ConnectionState::Connected(host)
        },
        ConnectionState::Connecting(_) if action == "fail" => {
            ConnectionState::Error("Connection refused")
        },
        ConnectionState::Connected(_) if action == "disconnect" => {
            ConnectionState::Disconnected
        },
        other => other
    }
}
```

9.13.4 The Option Pattern

A very common idiom in Lattice is using an `Option`-like enum to represent values that may or may not exist, and matching to handle both cases:

Listing 9.35: The Option pattern

```
enum Option {
    Some(any),
    None
}

fn find_user(id: Int) -> Option {
    if id == 1 {
        return Option::Some("Alice")
    }
    return Option::None
}

let user = find_user(1)

let greeting = match user {
    Option::Some(name) => "Hello, ${name}!",
    Option::None       => "User not found"
}

print(greeting) // Hello, Alice!
```

This pattern appears throughout Lattice code and is explored further in Chapter 10, where we combine it with `Result` types for robust error handling.

9.14 Exercises

1. **FizzBuzz with match.** Write a function `fn fizzbuzz(n: Int) -> String` that uses a `match` expression with guards to return `"FizzBuzz"` when `n` is divisible by both 3 and 5, `"Fizz"` when divisible by 3, `"Buzz"` when divisible by 5, and the number as a string otherwise. Hint: use a binding pattern with guards.

2. **List operations.** Write a function `fn sum_list(items: Array) -> Int` that computes the sum of an integer array using `match` with array destructuring and rest patterns. If the array is empty, return 0. If it has one element, return that element. If it has more, extract the first element and recursively sum the rest.

3. **Shape perimeter.** Define an enum `Shape` with variants `Circle(Float)`, `Rectangle(Float, Float)`, and `Square(Float)`. Write a function `fn perimeter(s: Shape) -> Float` that uses pattern matching to compute the perimeter. What happens if you forget to handle the `Square` variant?

4. **HTTP router.** Define a struct `Request` with fields `method: String` and `path: String`. Write a function `fn route(req: Request) -> String` that uses struct pattern matching to return different responses for `GET /`, `GET /about`, `POST /login`, and a catch-all `404` response.

5. **Expression evaluator.** Define an enum for arithmetic expressions:

```
enum Expr {
    Num(Int),
    Add(any, any),
    Mul(any, any)
}
```

   Write a function `fn evaluate(e: Expr) -> Int` that recursively evaluates an expression tree using pattern matching. For example, `Expr::Add(Expr::Num(2), Expr::Num(3))` should evaluate to 5.

What's Next Pattern matching gives us the tools to inspect data and branch on its structure. But what happens when things go wrong—when a file is missing, a network request times out, or a function receives unexpected input? In Chapter 10, we will explore Lattice's approach to error handling: the `error()` and `is_error()` functions, `try`/`catch` blocks, the `?` propagation operator, `defer` for cleanup, and how to design programs that fail gracefully.

Chapter 10

Error Handling

Things go wrong. Files disappear between the time you check for them and the time you open them. Network requests time out. Users type letters where numbers belong. A language that pretends errors do not happen is a language that produces programs crashing at 3 a.m. with no explanation.

Lattice takes errors seriously without making them painful. It gives you multiple tools—each suited to a different situation—and lets you choose the right one for the job. At the most basic level, the `error()` function creates error values that you can test with `is_error()`. When you need structured recovery, `try`/`catch` intercepts errors and hands you a detailed error map with a message, a line number, and a stack trace. For code that uses `Result` maps, the `?` operator unwraps successes and propagates failures in a single character. And `defer` ensures that cleanup code runs no matter how a scope exits—normally or through an error.

Listing 10.1: A quick tour of error handling

```
fn safe_divide(a: Float, b: Float) -> Map {
    if b == 0.0 {
        return err("division by zero")
    }
    return ok(a / b)
}

let result = safe_divide(10.0, 0.0)

let message = match result {
    {tag: "ok", value} => "Result: ${value}",
    {tag: "err", value} => "Error: ${value}"
}

print(message) // Error: division by zero
```

This chapter builds from the ground up. We start with the simplest error tools, progress through `try`/`catch` and the `?` operator, explore `defer` for resource cleanup, revisit contracts from Chapter 6, and finish with patterns for designing programs that fail gracefully.

10.1 Error Values: `error()` and `is_error()`

The simplest way to signal a problem in Lattice is to return an error value. The built-in `error()` function takes a string message and produces a special value that looks like a string but carries an error marker:

Listing 10.2: Creating and detecting error values

```
let bad = error("something went wrong")

print(is_error(bad))   // true
print(is_error("ok"))  // false
print(is_error(42))    // false
```

Under the hood, `error("msg")` returns a string prefixed with an internal marker (`EVAL_ERROR:`). When this marked string flows back to the runtime—for example, as a return value from a native function—the VM recognizes the prefix and routes it through the exception handler system. The `is_error()` function checks for this same prefix.

Listing 10.3: Using error() in a function

```
fn parse_int(s: String) -> any {
    let digits = "0123456789"
    for i in 0..s.len() {
        if !digits.contains(s.char_at(i)) {
            return error("invalid character: ${s.char_at(i)}")
        }
    }
    // ... parsing logic ...
    return 42 // simplified
}

let result = parse_int("12x4")

if is_error(result) {
    print("Parse failed")
} else {
    print("Got: ${result}")
}
// Parse failed
```

This approach works, but it has a limitation: the caller must remember to check `is_error()` on every return value. If you forget, the error value silently masquerades as a string. For more robust error handling, Lattice offers two better alternatives: the `Result` type (Section 10.2) and `try`/`catch` (Section 10.3).

10.2 The Result Type

Lattice's standard library (in `lib/fn.lat`) provides a convention for representing outcomes that might succeed or fail: the `Result` type. A Result is a plain `Map` with two keys:

- `"tag"`: either `"ok"` or `"err"`
- `"value"`: the success value or the error message

Two helper functions create Result maps:

Listing 10.4: Creating Result values

```
let success = ok(42)
let failure = err("file not found")

print(success) // {tag: ok, value: 42}
print(failure) // {tag: err, value: file not found}
```

Result Convention

A *Result* is a Map with a `"tag"` field set to `"ok"` or `"err"` and a `"value"` field carrying the payload. The standard library functions `ok(value)` and `err(message)` construct Result maps. The `?` operator and functions like `is_ok()`, `is_err()`, `unwrap()`, and `unwrap_or()` work with this convention.

10.2.1 Working with Results

The standard library provides several functions for inspecting and transforming Results:

Listing 10.5: Result helper functions

```
let result = ok(100)

print(is_ok(result))   // true
print(is_err(result))  // false

// Extract the value (crashes if err!)
print(unwrap(result))  // 100

// Extract with a default fallback
let r2 = err("timeout")
print(unwrap_or(r2, 0)) // 0
```

unwrap() on Errors

Calling `unwrap()` on an `err` Result triggers an assertion failure: `"unwrap() called on Err: ..."`. Use `unwrap()` only when you are certain the Result is `ok`, or prefer `unwrap_or()` for a safe fallback.

You can transform Results without unwrapping them using `map_result()` and `flat_map_result()`:

Listing 10.6: Transforming Results

```
let result = ok(5)

// Apply a function to the ok value
let doubled = map_result(result, |x| x * 2)
print(unwrap(doubled)) // 10

// Chain operations that return Results
let chained = flat_map_result(ok(10), |x| {
    if x > 100 {
        return err("too large")
    }
    return ok(x * x)
})
print(unwrap(chained)) // 100
```

`map_result()` applies a function to the value inside an `ok` and rewraps the result; if the input is `err`, it passes through unchanged. `flat_map_result()` does the same but expects the function to return a Result itself, avoiding double-wrapping.

10.3 try/catch

When an operation might throw a runtime error—a type mismatch, a division by zero, an assertion failure, or a contract violation—you can intercept it with `try`/`catch`:

Listing 10.7: Basic try/catch

```
let result = try {
    let x = 10 / 0
    x
} catch e {
    print("Caught: ${e.get("message")}")
    -1
}

print(result) // -1
```

10.3.1 try/catch Is an Expression

Like `if`/`else` and `match`, `try`/`catch` is an expression. The value of the `try` block (if it succeeds) or the `catch` block (if an error occurs) becomes the value of the entire expression:

Listing 10.8: try/catch produces a value

```
fn safe_get(items: Array, index: Int) -> any {
    return try {
        items[index]
    } catch e {
        nil
    }
}

print(safe_get([10, 20, 30], 1))  // 20
print(safe_get([10, 20, 30], 99)) // nil
```

10.3.2 The Error Map

The variable after `catch` (always required—you cannot omit it) is bound to a structured error `Map` with three fields:

Key	Type	Description
`"message"`	String	The error message
`"line"`	Int	Source line where the error occurred
`"stack"`	Array	Stack trace (array of strings)

Listing 10.9: Inspecting the error map

```
try {
    assert(false, "something broke")
} catch e {
    print(e.get("message")) // something broke
    print(e.get("line"))    // line number
    print(e.get("stack"))   // [<script> at line N]
}
```

The stack trace is an array of strings, each in the format `"function_name() at line N"`. For the top-level script, it appears as `"<script> at line N"`. For closures without names, it appears as `"<closure> at line N"`.

Listing 10.10: Reading the stack trace

```
fn inner() -> any {
    return error("deep error")
}

fn middle() -> any {
    return inner()
}

try {
    middle()
} catch e {
    let trace = e.get("stack")
    for entry in trace {
        print("  ${entry}")
    }
}
// Prints the call chain from inner() through middle() to <script>
```

10.3.3 What Can Be Caught?

The `try`/`catch` mechanism catches a wide range of runtime errors:

- Division by zero
- Index out of bounds
- Type mismatches (parameter and return type checks)
- Assertion failures (`assert()`, `assert_eq()`, etc.)
- Contract violations (`require` and `ensure`)
- `error()` values routed through the runtime
- Freeze and anneal contract failures
- Missing module exports

Under the hood, `try` compiles to `OP_PUSH_EXCEPTION_HANDLER`, which registers a handler with the VM. The handler records the catch block's instruction pointer, the current call frame index, and a snapshot of the stack top. When an error fires (via the `VM_ERROR` macro or `OP_THROW`), the VM looks for the nearest handler, unwinds the call stack to the handler's frame, restores the stack, and jumps to the catch block with the error map pushed on top. You can see this mechanism in `src/stackvm.c`.

Handler Limit

The VM supports up to 64 nested exception handlers (`STACKVM_HANDLER_MAX`). This is more than enough for any reasonable program. If you exceed this limit, the VM throws its own error: `"too many nested exception handlers"`.

10.4 The ? Operator

The ? operator is Lattice's most concise error-handling tool. It works with Result maps (Section 10.2): if the Result is `ok`, the ? extracts the inner value; if it is `err`, it immediately returns the error from the current function.

Listing 10.11: The ? operator in action

```
fn read_config(path: String) -> Map {
    // Imagine these return Results
    let content = read_file(path)?   // propagates err
    let parsed = parse_json(content)? // propagates err
    return ok(parsed)
}
```

Without ?, you would need to manually check each Result:

Listing 10.12: Manual error checking (the verbose way)

```
fn read_config(path: String) -> Map {
    let content_result = read_file(path)
    if is_err(content_result) {
        return content_result
    }
    let content = unwrap(content_result)

    let parsed_result = parse_json(content)
    if is_err(parsed_result) {
        return parsed_result
    }
    return ok(unwrap(parsed_result))
}
```

The ? operator compresses each three-line check-and-unwrap into a single postfix character.

10.4.1 How ? Works

At compile time, `expr?` compiles to the expression followed by a single `OP_TRY_UNWRAP` opcode. At runtime, this opcode inspects the top of the stack:

1. If the value is a Map with `"tag"` = `"ok"`: extract the `"value"` field and replace the stack top with it. Execution continues normally.
2. If the value is a Map with `"tag"` = `"err"`: immediately return from the current function with the error map as the return value. No exception handler is needed; the operator uses the normal return path.
3. If the value is not a conforming Result map: throw a runtime error: `"'?' operator requires a result map with \{tag: \"ok\"` ërr; value: ...}"|.

? Requires Result Maps

The ? operator only works with maps that follow the Result convention (`ok()`/`err()` maps). Using ? on a plain integer, string, or non-conforming map will throw a runtime error. If you are working with `error()` values instead of Result maps, use `is_error()` or `try`/`catch` instead.

10.4.2 Chaining ?

Because ? unwraps the `ok` value in place, you can chain it with method calls and further operations:

Listing 10.13: Chaining the ? operator

```
fn process_data(input: Map) -> Map {
    let validated = validate(input)?
    let transformed = transform(validated)?
    let saved = save_to_db(transformed)?
    return ok("Processed ${saved} records")
}
```

Each ? either continues with the unwrapped value or short-circuits the entire function, returning the first error encountered. This creates a clean, linear flow of operations where errors are handled implicitly.

10.5 panic()

When something goes so catastrophically wrong that recovery makes no sense, `panic()` signals a fatal error:

Listing 10.14: Using panic() for unrecoverable errors

```
fn initialize(config: Map) -> any {
    let db_url = config.get("database_url")
    if db_url == nil {
        panic("Cannot start without a database URL")
    }
    return db_url
}
```

`panic()` is semantically different from `error()`: it communicates that the program has reached a state that should never happen—an invariant violation, a logic error, or a missing critical resource. Use `error()` and Result types for expected failures (network timeouts, invalid user input); reserve `panic()` for bugs and impossible situations.

When to panic()

Good uses of `panic()`: unreachable code branches, failed internal invariants, missing required configuration at startup. Bad uses: user input validation, file-not-found errors, network failures—these are expected and should be handled with Results or `try`/`catch`.

10.6 defer

Resources need cleanup. Files should be closed, locks released, temporary directories removed. The problem with manual cleanup is that errors can cause early exits, skipping the cleanup code. `defer` solves this by guaranteeing that a block of code runs when the enclosing scope exits, no matter how it exits:

Listing 10.15: defer ensures cleanup runs

```
fn process_file(path: String) -> any {
    let handle = open_file(path)
    defer {
        close_file(handle)
        print("File closed")
    }

    // Even if this throws, the defer block runs
    let data = read_all(handle)
    return parse(data)
}
```

10.6.1 Execution Order: LIFO

When multiple `defer` blocks are registered in the same scope, they execute in *last-in, first-out* (LIFO) order—the most recently deferred block runs first:

Listing 10.16: Defers run in reverse order

```
fn demo() -> any {
    defer { print("First deferred, runs last") }
    defer { print("Second deferred, runs first") }
    print("Function body")
    return nil
}

demo()
// Function body
// Second deferred, runs first
// First deferred, runs last
```

This LIFO ordering is intentional: resources acquired later typically depend on resources acquired earlier, so they should be released first.

10.6.2 Scope-Aware Execution

Defers are tied to their enclosing scope. When a scope exits—whether it is a function body, a block, or a loop iteration—only the defers registered in that scope run:

Listing 10.17: Defer and scope boundaries

```
fn outer() -> any {
    defer { print("outer defer") }

    for i in 0..3 {
        defer { print("loop defer ${i}") }
        print("iteration ${i}")
    }

    print("after loop")
    return nil
}

outer()
// iteration 0
// loop defer 0
// iteration 1
// loop defer 1
// iteration 2
// loop defer 2
// after loop
// outer defer
```

Each loop iteration creates and destroys a scope, so the loop's defer runs at the end of each iteration, not at the end of the function.

10.6.3 Defer and Errors

Defers run even when an error causes an early exit. This is the whole point—cleanup that only runs on the happy path is not reliable cleanup:

Listing 10.18: Defer runs on error paths

```
fn risky() -> any {
    defer { print("cleanup always runs") }

    print("about to fail")
    assert(false, "intentional failure")
    print("this never prints")
    return nil
}

try {
    risky()
} catch e {
    print("caught: ${e.get("message")}")
}
// about to fail
// cleanup always runs
// caught: intentional failure
```

10.6.4 Defer and Return Values

An important detail: deferred blocks do not affect the return value of the enclosing function. The return value is determined *before* defers execute, and it is preserved across all defer block executions:

Listing 10.19: Defer does not alter return values

```
fn compute() -> Int {
    defer { print("cleaning up") }
    return 42
}

let result = compute()
// cleaning up
print(result) // 42
```

Under the hood, when `OP_DEFER_RUN` executes, the VM saves the current top-of-stack value (the return value), runs each defer body, then restores the saved value. Defer bodies share the parent frame's local variables, so they can read (and even modify) locals, but they cannot change what the function returns.

10.6.5 How Defer Compiles

The defer body is compiled inline in the function's bytecode. An `OP_DEFER_PUSH` instruction records the body's location and scope depth, then a jump skips past the body during normal execution. When `OP_DEFER_RUN` fires (at scope exit or function return), the VM creates a lightweight wrapper and executes the body as a sub-frame.

At every return point in a compiled function, the compiler emits the sequence: (1) return type check, (2) ensure postcondition checks, (3) `OP_DEFER_RUN` with scope depth 0 (run all defers), (4) `OP_RETURN`. This guarantees defers run regardless of which return path the function takes.

10.7 Contracts Revisited

In Chapter 6 we introduced `require` and `ensure` contracts. Now that we understand error handling, let us see how they fit into the bigger picture.

10.7.1 require: Preconditions

A `require` contract checks a condition at function entry. If the condition is false, it throws an error that can be caught by `try`/`catch`:

Listing 10.20: require throws catchable errors

```
fn withdraw(balance: Float, amount: Float) -> Float
    require amount > 0.0, "amount must be positive"
    require amount <= balance, "insufficient funds"
{
    return balance - amount
}

// Successful call
print(withdraw(100.0, 30.0)) // 70.0

// Failed precondition
try {
    withdraw(100.0, -5.0)
} catch e {
    print(e.get("message"))
    // require failed in 'withdraw': amount must be positive
}
```

The error message always includes the function name and the custom message (or `"condition not met"` if no message was provided). Under the hood, the compiler emits the condition expression followed by `OP_JUMP_IF_TRUE` (skip if ok) or `OP_THROW` with the formatted message string.

10.7.2 ensure: Postconditions

An `ensure` contract checks the return value of a function. It takes a closure that receives the return value and must return a truthy value:

Listing 10.21: ensure validates return values

```
fn half(n: Int) -> Int
    ensure |result| { result * 2 == n }, "result must be half of input"
{
    return n / 2
}

print(half(10)) // 5

// With an odd number, integer division truncates
try {
    half(7)
} catch e {
    print(e.get("message"))
    // ensure failed in 'half': result must be half of input
}
```

The ensure check runs at every return point—both explicit `return` statements and the implicit trailing expression. The compiler inserts the check *before* defers run, so the order at each return point is: type check → ensure checks → defer execution → actual return.

10.7.3 Combining Contracts with try/catch

Because contract violations throw catchable errors, you can use `try`/`catch` to handle them gracefully:

Listing 10.22: Catching contract violations

```
fn positive_sqrt(n: Float) -> Float
    require n >= 0.0, "cannot take sqrt of negative number"
{
    // Newton's method approximation
    flux guess = n / 2.0
    for _ in 0..20 {
        guess = (guess + n / guess) / 2.0
    }
    return guess
}

let values = [4.0, -1.0, 9.0, -16.0]

for v in values {
    let result = try {
        positive_sqrt(v)
    } catch e {
        e.get("message")
    }
    print("sqrt(${v}) = ${result}")
}
// sqrt(4.0) = 2.0
// sqrt(-1.0) = require failed in 'positive_sqrt': ...
// sqrt(9.0) = 3.0
// sqrt(-16.0) = require failed in 'positive_sqrt': ...
```

10.8 Assertions

Lattice provides a family of assertion functions for testing invariants:

Function	Purpose
`assert(cond)`	Fails if `cond` is falsy
`assert(cond, msg)`	Fails with custom message
`assert_eq(a, b)`	Fails if `a != b`
`assert_ne(a, b)`	Fails if `a == b`
`assert_true(val)`	Fails if `val` is not `true`
`assert_false(val)`	Fails if `val` is not `false`
`assert_nil(val)`	Fails if `val` is not `nil`
`assert_contains(s, sub)`	Fails if `s` does not contain `sub`
`assert_type(val, type)`	Fails if `val` is not of type `type`
`assert_throws(closure)`	Fails if closure does *not* throw

All assertion failures are catchable with `try`/`catch`. The `assert_throws()` function is particularly useful in tests—it calls a closure and succeeds only if the closure throws:

Listing 10.23: Testing that code throws

```
test "division by zero throws" {
    assert_throws(|| {
        let x = 1 / 0
    })
}

test "require contracts throw on violation" {
    fn positive(n: Int) -> Int
        require n > 0, "must be positive"
    {
        return n
    }

    assert_throws(|| { positive(-1) })
}
```

10.9 The `try_fn` Helper

The standard library provides `try_fn()`, which bridges the gap between `try`/`catch` exceptions and the `Result` type. It runs a closure inside a `try`/`catch` and returns the outcome as a Result:

Listing 10.24: try_fn converts exceptions to Results

```
let result = try_fn(|| {
    let x = 10 / 0
    x
})

print(is_err(result))         // true
print(result.get("value"))    // error map with message, line, stack
```

This is useful when you want to call a function that might throw but you prefer to work with Result maps rather than `try`/`catch` blocks:

Listing 10.25: Using try_fn with the ? operator

```
fn safe_pipeline(data: any) -> Map {
    let step1 = try_fn(|| { validate(data) })?
    let step2 = try_fn(|| { transform(step1) })?
    return ok(step2)
}
```

10.10 Designing for Graceful Failure

Now that we have seen all the error handling tools, let us discuss when to use each one.

10.10.1 Choosing the Right Tool

Situation	Recommended Approach
Expected failures (parsing, I/O, validation)	Return `Result` maps with `ok()`/`err()`
Calling code that throws	Wrap in `try`/`catch` or `try_fn()`
Chaining fallible operations	Use ? operator with Results
Bugs and impossible states	`panic()` or `assert()`
Function input validation	`require` contracts
Function output guarantees	`ensure` contracts
Resource cleanup	`defer`

10.10.2 The Error Boundary Pattern

A common architecture is to let errors propagate freely through the interior of your program and catch them at well-defined *boundaries*—the top of a request handler, the main loop of a CLI tool, the entry point of a worker:

Listing 10.26: The error boundary pattern

```
fn handle_request(req: any) -> String {
    // Interior code uses ? and Results freely
    let result = try {
        let user = authenticate(req)?
        let data = fetch_data(user)?
        let response = format_response(data)?
        unwrap(response)
    } catch e {
        // Boundary: convert any error to a user-friendly response
        let msg = e.get("message")
        print("Error handling request: ${msg}")
        "500 Internal Server Error"
    }
    return result
}
```

Interior functions return Results and use `?` to propagate errors upward. The boundary catches everything, logs the details, and returns a safe response to the caller.

10.10.3 Fail Fast, Recover High

A principle that serves Lattice programs well: **fail fast at the point of failure, recover at the point where you have enough context to do something useful.**

Do not catch errors deep inside helper functions just to re-throw them. Let them propagate. Catch them where you can take meaningful action—retry, return a default, log and continue, or shut down cleanly.

Listing 10.27: Let errors propagate to where context exists

```
// Bad: catching too early
fn get_user_name(id: Int) -> String {
    let result = try {
        fetch_user(id)
    } catch e {
        // We don't know what to do here!
        return "Unknown"
    }
    return result.get("name")
}

// Better: propagate and let the caller decide
fn get_user_name(id: Int) -> Map {
    let user = fetch_user(id)?
    return ok(user.get("name"))
}

// Caller has context to handle the error
fn display_profile(id: Int) -> String {
    let name = match get_user_name(id) {
        {tag: "ok", value} => value,
        {tag: "err", value} => "Guest"
    }
    return "Welcome, ${name}!"
}
```

10.11 Exercises

1. **Safe division.** Write a function `fn safe_div(a: Float, b: Float) -> Map` that returns `ok(a / b)` when `b` is not zero and `err("division by zero")` otherwise. Then write a function `fn chain_divide(values: Array) -> Map` that divides the first element by each subsequent element in sequence, using `?` to propagate errors.

2. **Resource manager.** Write a function that simulates opening three resources (use print statements to show "opened resource N"). Use three `defer` blocks to close them. Introduce an error after opening the second resource and verify that all opened resources are still closed.

3. **Retry logic.** Write a function `fn retry(attempts: Int, action: any) -> Map` that calls the `action` closure up to `attempts` times. If the action succeeds (returns without throwing), return

`ok(result)`. If all attempts throw, return `err("all attempts failed")`. Use `try`/`catch` inside the retry loop.

4. **Contract design.** Write a function `fn clamp(value: Int, low: Int, high: Int) -> Int` with a `require` contract that ensures `low <= high`, and an `ensure` contract that verifies the result is within the `[low, high]` range. Test it with values inside and outside the range, and verify that the contracts catch violations.

5. **Pipeline with errors.** Build a data processing pipeline with three stages—validate, transform, and format—each returning a Result. Write a `fn pipeline(input: any) -> Map` that chains the three stages using `?`. Test it with inputs that fail at each stage and verify that the correct error is returned.

What's Next With control flow, functions, collections, strings, pattern matching, and error handling in your toolkit, you are equipped to write substantial Lattice programs. But we have been working with values that are either immutable by default or mutable by declaration without fully understanding *why*. In Part III, we dive deep into the *phase system*—Lattice's most distinctive feature. Chapter 11 reveals how the fluid, crystal, and sublimated phases govern mutability, how freezing and thawing work at the memory level, and why thinking of values as materials with physical states leads to safer, more predictable programs.

Part III

The Phase System

Chapter 11

Phases Explained

Every material in nature exists in a state. Water flows as liquid, hardens as ice, disperses as steam. The state determines what you can *do* with the material—you can pour water but not ice, you can stack ice but not steam. Lattice borrows this intuition and applies it to data: every value in the language carries a *phase* that governs whether it can be changed, shared, or locked down permanently.

If you have been following along from **??**, you already know the basics: `flux` for mutable, `fix` for immutable, and `let` for "figure it out for me." This chapter goes deeper. We will explore the full philosophy behind the phase system, examine each phase keyword in detail, master the trio of `freeze()`, `thaw()`, and `clone()`, learn when and why to freeze values, and understand the error messages the phase checker produces when something goes wrong.

11.1 The Philosophy: Mutability as a Material Property

Most languages treat mutability as a *declaration modifier*. In Rust, you write `let mut`. In JavaScript, you choose between `let` and `const`. In both cases, the declaration tells the *binding* whether it can be reassigned, but the underlying data may or may not follow suit.

Lattice takes a different stance. Mutability is not a property of the *binding*—it is a property of the *value itself*. When you declare a variable with `flux`, the value stored inside is tagged as *fluid*: it lives in the mutable heap, the garbage collector watches over it, and you can modify it freely. When you declare with `fix`, the value is tagged as *crystal*: it is hardened, immutable, and may be stored in a memory arena where it can never be altered.

Phase

A *phase* is a runtime tag on every Lattice value that describes its mutability state. There are four phase tags in the runtime:

- **Fluid** (`VTAG_FLUID`) — mutable, alive, in motion.
- **Crystal** (`VTAG_CRYSTAL`) — immutable, hardened, permanent.
- **Unphased** (`VTAG_UNPHASED`) — newly created, not yet committed to either side.
- **Sublimated** (`VTAG_SUBLIMATED`) — permanently immutable; cannot be thawed back.

This design has a profound consequence: you can take a fluid value and *freeze* it into a crystal. You can take a crystal and *thaw* it back into a fluid. You can *clone* a value to get an independent copy without changing the original's phase. The phase is the material property of the data itself, and Lattice gives you the tools to change that material state deliberately.

Why does this matter? Consider a configuration object. While your application is starting up, the configuration needs to be mutable—you are loading defaults, overriding with environment variables, parsing command-line flags. But once startup is complete, the configuration should be frozen solid: no accidental mutation, no race conditions, no surprises. In most languages, you enforce this with discipline and documentation. In Lattice, you enforce it with a single call to `freeze()`.

Listing 11.1: Freezing a configuration after setup

```
flux config = Map::new()
config.set("host", "0.0.0.0")
config.set("port", "8080")
config.set("debug", "false")

// Startup complete. Lock it down.
fix app_config = freeze(config)
print(phase_of(app_config))  // "crystal"
```

After the freeze, `app_config` is crystal. Any attempt to call `app_config.set("host", "evil.com")` will fail with a phase error at runtime. The data is safe—not because of a linting rule or a code review comment, but because the material itself has hardened.

11.2 `flux` (Fluid) — Mutable, Alive, in Motion

The `flux` keyword declares a binding whose value is in the *fluid* phase. Fluid values can be reassigned, mutated in place, grown, shrunk, and generally treated as living, changing data.

Listing 11.2: Basic `flux` usage

```
flux counter = 0
counter = counter + 1
counter += 1
print(counter)  // 2
```

When you create a value with `flux`, the runtime tags it as `VTAG_FLUID`. Under the hood (in `include/value.h`), every `LatValue` struct carries a `phase` field of type `PhaseTag`. A fluid value's phase tag is set to `VTAG_FLUID`, and it lives on the *FluidHeap*—the garbage-collected portion of memory managed by the mark-sweep collector.

11.2.1 Fluid Collections

Fluid values are not limited to scalars. Arrays, maps, sets, and structs can all be fluid:

Listing 11.3: Fluid collections

```
flux temperatures = [18.5, 19.2, 20.1]
temperatures.push(21.0)
temperatures.push(22.3)
print(temperatures)  // [18.5, 19.2, 20.1, 21.0, 22.3]

flux users = Map::new()
users.set("alice", 42)
users.set("bob", 37)
print(users.len())  // 2
```

When a compound value like an array is fluid, all of its elements inherit the fluid phase. You can modify the array and its contents freely.

11.2.2 Fluid Reassignment

Because `flux` values are mutable bindings, you can reassign them entirely:

Listing 11.4: Reassigning a `flux` binding

```
flux greeting = "hello"
print(phase_of(greeting))  // "fluid"

greeting = "bonjour"
print(greeting)  // "bonjour"

// Even change the type entirely
greeting = 42
print(greeting)  // 42
```

Lattice is dynamically typed, so a `flux` binding can hold any value. The phase stays fluid unless you explicitly transition it.

Fluid Does Not Mean Unprotected

A fluid value is mutable, but it is still tracked by the runtime. The garbage collector manages its memory, the phase checker watches for incorrect transitions, and in strict mode, the compiler verifies that you do not accidentally pass a fluid value where a crystal is expected.

11.3 `fix` (Crystal) — Immutable, Hardened, Permanent

The `fix` keyword declares a binding whose value is in the *crystal* phase. Crystal values are immutable: they cannot be reassigned, their internal state cannot be modified, and any attempt to mutate them produces a runtime error.

Listing 11.5: Basic `fix` usage

```
fix pi = 3.14159
fix greeting = "Hello, Lattice!"
fix primes = freeze([2, 3, 5, 7, 11])

print(pi)        // 3.14159
print(greeting)  // Hello, Lattice!
print(primes)    // [2, 3, 5, 7, 11]
```

Scalars with `fix`

Scalar values (integers, floats, booleans, strings) bound with `fix` are automatically crystal. You do not need to call `freeze()` on them—the runtime handles the phase assignment. For compound values like arrays, maps, and structs, you typically call `freeze()` explicitly to ensure the deep crystallization (more on this in Section 11.5).

11.3.1 Why Crystal Is Not Just "const"

In many languages, `const` merely prevents *rebinding*—you cannot assign a new value to the variable, but you can still modify the data it points to. JavaScript's `const` is the canonical example:

Listing 11.6: The problem with shallow const (JavaScript-style)

```
// In JavaScript:
// const arr = [1, 2, 3];
// arr.push(4);  // This works! const doesn't protect contents.

// In Lattice:
fix arr = freeze([1, 2, 3])
// arr.push(4)  // Runtime error: cannot mutate crystal value
```

Lattice's crystal phase is *deep*. When you freeze an array, every element becomes crystal. When you freeze a struct, every field becomes crystal. When you freeze a map, every key-value pair becomes crystal. The immutability propagates through the entire object graph.

Under the hood, the `value_freeze()` function in `src/value.c` calls `set_phase_recursive()`, which walks the entire value tree and sets every nested value's phase to `VTAG_CRYSTAL`:

Listing 11.7: Deep freeze in action

```
flux nested = [[1, 2], [3, 4], [5, 6]]
fix frozen = freeze(nested)

print(phase_of(frozen))  // "crystal"
// Every inner array is also crystal---you cannot push
// into frozen[0] any more than you can push into frozen itself.
```

11.3.2 Crystal Values and Memory

Crystal values may be relocated from the FluidHeap into a *CrystalRegion*—an arena-backed memory area that provides cache locality and O(1) bulk deallocation. We explore this memory architecture in detail in Chapter 12. For now, the key insight is that crystal values are not merely "values you are not allowed to change." They are values that *the runtime treats differently at the memory level*, storing them in a region optimized for immutable data.

11.4 `let` — Phase Inference

Not every binding needs an explicit phase declaration. The `let` keyword tells Lattice: "I do not care about the phase right now—infer it from context."

Listing 11.8: `let` bindings with inferred phase

```
let name = "Lattice"
let count = 42
let items = [1, 2, 3]

print(phase_of(name))   // "unphased"
print(phase_of(count))  // "unphased"
print(phase_of(items))  // "unphased"
```

When you use `let`, the value starts in the `VTAG_UNPHASED` state. Unphased values are not locked into either fluid or crystal. They behave as fluid in practice—you can reassign and mutate them—but they do not carry the explicit guarantee of either phase.

Unphased

An *unphased* value (`VTAG_UNPHASED`) has no explicit phase commitment. It can be treated as mutable and can transition to either fluid or crystal through explicit operations. In casual mode (the default), `let` bindings are unphased. In strict mode, `let` is *forbidden*—you must choose `flux` or `fix` explicitly.

The phase checker in `src/phase_check.c` handles `let` differently depending on the execution mode. In casual mode (`MODE_CASUAL`), the effective phase of a `let` binding is inferred from the initializer expression:

Listing 11.9: Phase inference with `let`

```
let frozen_data = freeze([1, 2, 3])
// frozen_data's effective phase is crystal (inferred from freeze())

let thawed_copy = thaw(frozen_data)
// thawed_copy's effective phase is fluid (inferred from thaw())
```

When to Use `let`

Use `let` for quick prototyping and REPL exploration. When writing production code, prefer `flux` or `fix` to make your intent explicit. In strict mode, you are *required* to make this choice—the compiler will refuse `let` bindings entirely.

11.4.1 The Relationship Between `let` and Strict Mode

In strict mode (`#mode strict`), the phase checker rejects `let` bindings with a clear error:

Listing 11.10: `let` rejected in strict mode

```
#mode strict

let name = "Lattice"
// Error: strict mode: use 'flux' or 'fix' instead of 'let'
//        for binding 'name'
```

This is by design. Strict mode exists for codebases where phase discipline matters—library code, concurrent systems, safety-critical paths. We cover strict mode in full detail in Chapter 14.

11.5 `freeze()`, `thaw()`, `clone()` in Depth

These three built-in functions are the tools you use to change a value's phase or create independent copies. Think of them as the laboratory equipment for Lattice's material science.

11.5.1 `freeze()` — Crystallization

`freeze()` takes a value and returns a crystal copy. The original fluid value is unaffected (though it may be freed by garbage collection if no longer referenced), and the returned value is deeply immutable.

Listing 11.11: The `freeze()` function

```
flux temperatures = [20.1, 21.5, 19.8, 22.0]
fix snapshot = freeze(temperatures)

print(phase_of(temperatures))  // "fluid"
print(phase_of(snapshot))      // "crystal"

// The fluid value is still alive and mutable
temperatures.push(23.1)
print(temperatures)  // [20.1, 21.5, 19.8, 22.0, 23.1]

// The crystal snapshot is frozen in time
print(snapshot)  // [20.1, 21.5, 19.8, 22.0]
```

Under the hood, `freeze()` does two things:

1. **Deep clones** the value into a new `CrystalRegion` (arena-backed memory). This is performed by `value_deep_clone()` in `src/value.c`, which recursively copies every nested value.

2. **Sets the phase tag** to `VTAG_CRYSTAL` on every value in the tree via the `set_phase_recursive()` function.

The deep clone ensures that the crystal value has no pointers back into the FluidHeap. This is a critical invariant: crystal values must be completely independent of the garbage-collected heap so that the GC can skip them during mark-sweep cycles.

Freeze Is Not Free

`freeze()` performs a deep copy. For large nested structures, this has a real cost in both time and memory. However, this cost buys you a powerful guarantee: the frozen value is truly independent and will never be mutated, even if the original value continues to change.

Freezing with Contracts

`freeze()` can take an optional contract closure that validates the value before crystallization:

Listing 11.12: Freeze with contract validation

```
flux score = 85

// freeze with a validation contract
fix validated = freeze(score, |val: any| {
    if val < 0 { return "score must be non-negative" }
    if val > 100 { return "score must not exceed 100" }
    return nil
})

print(validated)  // 85
```

If the contract returns a non-nil value, the freeze fails and the error message is propagated. This pattern lets you enforce invariants at the moment of crystallization—the value only hardens if it passes validation.

11.5.2 thaw() — Melting a Crystal

thaw() is the inverse of freeze(). It takes a crystal value and returns a *new* fluid copy. The original crystal value remains unchanged and immutable.

Listing 11.13: The thaw() function

```
fix frozen_list = freeze([10, 20, 30])
print(phase_of(frozen_list))  // "crystal"

flux thawed = thaw(frozen_list)
print(phase_of(thawed))  // "fluid"

// Now we can modify the thawed copy
thawed.push(40)
print(thawed)       // [10, 20, 30, 40]
print(frozen_list)  // [10, 20, 30]  -- unchanged
```

The implementation in src/value.c reveals an important detail: value_thaw() first calls value_deep_clone() to create an independent copy, then calls set_phase_recursive() to set the phase to VTAG_FLUID. This means thaw() always creates a new copy—it never modifies the original crystal value in place.

Thaw Creates a Copy

A common misconception is that `thaw()` "unfreezes" the original value. It does not. `thaw()` creates a brand-new fluid copy, leaving the original crystal intact. If you need to modify a frozen value, thaw it, modify the copy, and optionally re-freeze it.

Thawing Refs

When `thaw()` operates on a `Ref` value (a reference-counted shared wrapper), it breaks the sharing. The thawed result is a new `Ref` with a deep-cloned inner value, independent of any other references to the original. This is documented in `src/value.c` in the `value_thaw()` function:

Listing 11.14: Thawing a Ref breaks sharing

```
fix shared_ref = freeze(Ref(42))
flux independent = thaw(shared_ref)
// independent is a new Ref, no longer sharing data with shared_ref
```

11.5.3 `clone()` — Independent Copy

`clone()` creates a deep copy of a value *without changing its phase*. A fluid value cloned stays fluid. A crystal value cloned stays crystal. The result is a structurally identical but completely independent copy.

Listing 11.15: The `clone()` function

```
flux original = [1, 2, 3]
flux copy = clone(original)

copy.push(4)
print(original)  // [1, 2, 3]  -- unaffected
print(copy)      // [1, 2, 3, 4]

print(phase_of(original))  // "fluid"
print(phase_of(copy))      // "fluid"  -- same phase as original
```

Clone is especially useful when you want to pass a value to a function that might modify it, but you want to preserve the original:

Listing 11.16: Cloning for safe function calls

```
fn process_data(data: any) {
    // This function modifies its argument
    data.push(0)
    return data
}

flux sensor_readings = [45.2, 47.8, 46.1]
flux processed = process_data(clone(sensor_readings))

print(sensor_readings)  // [45.2, 47.8, 46.1]  -- safe
print(processed)        // [45.2, 47.8, 46.1, 0]
```

When to Use Each

- Use `freeze()` when you want to lock a value down for safety, sharing, or performance.
- Use `thaw()` when you need to modify data that is currently frozen.
- Use `clone()` when you want an independent copy without changing the phase.

11.6 Advanced Phase Operations

Beyond the fundamental trio, Lattice offers several advanced phase operations for fine-grained control over the crystallization lifecycle.

11.6.1 `forge` — Scoped Construction

A `forge` block lets you construct a complex crystal value through a series of mutable operations, automatically freezing the result when the block completes. Think of it like a foundry: the material is molten inside the forge, but what comes out is solid.

Listing 11.17: Building a crystal value with `forge`

```
fix server_config = forge {
    flux temp = Map::new()
    temp.set("host", "localhost")
    temp.set("port", "8080")
    temp.set("max_connections", "1000")
    temp.set("timeout_ms", "30000")
    freeze(temp)
}

print(phase_of(server_config))  // "crystal"
print(server_config.get("host"))  // "localhost"
```

The `forge` block is especially valuable when the construction process requires loops, conditionals, or function calls:

Listing 11.18: Complex construction inside `forge`

```
fix lookup_table = forge {
    flux table = Map::new()
    for i in 0..100 {
        table.set(to_string(i), i * i)
    }
    freeze(table)
}

print(lookup_table.get("7"))   // 49
print(lookup_table.get("12"))  // 144
```

The phase checker knows that a `forge` block always produces a crystal result. This is reflected in `src/phase_check.c`, where `EXPR_FORGE` returns `PHASE_CRYSTAL`.

11.6.2 `anneal` — Transform a Crystal

In metallurgy, *annealing* is the process of heating a metal, transforming it, and letting it cool back to a hardened state. Lattice's `anneal` does the same: it thaws a crystal value, applies a transformation closure, and re-freezes the result.

Listing 11.19: Annealing a crystal value

```
fix prices = freeze([9.99, 14.99, 19.99, 24.99])

// Apply a 10% discount without manually thawing/re-freezing
fix discounted = anneal(prices, |items: any| {
    flux result = []
    for price in items {
        result.push(price * 0.9)
    }
    return result
})

print(discounted)             // [8.991, 13.491, 17.991, 22.491]
print(phase_of(discounted))   // "crystal"
```

`anneal` requires its first argument to be a crystal value. If you pass a fluid value, the runtime produces an error. The transformation closure receives the thawed (fluid) copy, and whatever it returns is automatically frozen.

11.6.3 `crystallize` and `borrow` — Scoped Phase Changes

Sometimes you want to temporarily change a value's phase within a specific scope. `crystallize` temporarily freezes a variable for the duration of a block, then restores its original phase:

Listing 11.20: Temporary crystallization

```
flux data = [1, 2, 3]

crystallize(data) {
    // Inside this block, data is crystal
    print(phase_of(data))  // "crystal"
    // data.push(4)  // This would error!
}

// Back to fluid outside
print(phase_of(data))  // "fluid"
data.push(4)
print(data)  // [1, 2, 3, 4]
```

`borrow` is the inverse: it temporarily thaws a crystal variable so you can modify it within a block, then re-freezes it:

Listing 11.21: Temporarily borrowing a crystal value

```
fix config = freeze(Map::new())

borrow(config) {
    // Inside this block, config is fluid
    config.set("debug", "true")
    config.set("verbose", "true")
}

// config is crystal again outside the borrow block
print(phase_of(config))  // "crystal"
```

These scoped operations are compiled to a sequence of phase tag changes and are verified by the phase checker (see `src/stackcompiler.c`). The key property: both `crystallize` and `borrow` restore the original phase when the block ends, even if an error occurs during execution.

11.6.4 `sublimate` — The Point of No Return

In chemistry, sublimation is when a solid transforms directly into gas, skipping the liquid state entirely. In Lattice, `sublimate` converts a value into the *sublimated* phase (`VTAG_SUBLIMATED`)—a terminal, irreversible state of immutability.

Listing 11.22: Sublimating a value

```
flux secret_key = "my-api-key-12345"
sublimate(secret_key)

print(phase_of(secret_key))  // "sublimated"
// secret_key cannot be thawed, reassigned, or modified
// It is permanently locked
```

Unlike crystal values, sublimated values *cannot be thawed*. Calling `thaw()` on a sublimated value produces an error. This makes sublimation ideal for security-sensitive data (API keys, passwords, certificates) or constants that must never change under any circumstances.

Sublimation Is Irreversible

Once a value is sublimated, there is no way to make it mutable again within the runtime. Use sublimation deliberately for values that truly must be permanent.

11.7 When and Why to Freeze Values

Knowing *how* to freeze is only half the story. Knowing *when* to freeze is what separates a novice Lattice programmer from an expert. Here are the primary scenarios where freezing pays dividends.

11.7.1 Sharing Data Between Threads

In Lattice's structured concurrency model (covered in Chapter 19), spawned tasks should not share mutable state. The strict mode phase checker enforces this: fluid bindings cannot cross a spawn boundary. The solution is to freeze data before sharing:

Listing 11.23: Freezing data for safe sharing across tasks

```
flux results = [10, 20, 30, 40, 50]
fix shared = freeze(results)

scope {
    spawn {
        // Safe: shared is crystal, no data races
        print("Task sees: " + to_string(shared))
    }
    spawn {
        print("Other task sees: " + to_string(shared))
    }
}
```

Crystal values are inherently safe to share because no one can modify them. This eliminates an entire class of concurrency bugs—no locks needed, no synchronization overhead, no data races.

11.7.2 API Boundaries

When designing a function that returns data the caller should not modify, freeze it:

Listing 11.24: Returning frozen data from a function

```
fn get_default_settings() {
    flux defaults = Map::new()
    defaults.set("theme", "dark")
    defaults.set("font_size", "14")
    defaults.set("language", "en")
    return freeze(defaults)
}

fix settings = get_default_settings()
// Callers get crystal data---they can read but not modify
```

11.7.3 Snapshot Semantics

When you need to capture the state of a value at a particular moment, freeze it:

Listing 11.25: Creating snapshots with freeze

```
flux log_entries = []
log_entries.push("Server started")
log_entries.push("Listening on port 8080")

fix startup_log = freeze(log_entries)

log_entries.push("Connection from 192.168.1.1")
log_entries.push("Request: GET /api/status")

fix runtime_log = freeze(log_entries)

print(startup_log.len())  // 2
print(runtime_log.len())  // 4
```

Each `freeze()` call captures the array at that point in time. The snapshots are independent and immutable, while the original continues to grow.

11.7.4 Performance Optimization

Crystal values stored in arena-backed CrystalRegions enjoy better cache locality than fluid values scattered across the general heap. For read-heavy workloads with large data structures, freezing the data can improve access performance. We explore the memory implications in Chapter 12.

11.7.5 Correctness Guarantees

Sometimes you freeze a value simply to catch bugs. If you know a value should not change after a certain point, freezing it converts silent corruption into a loud runtime error:

Listing 11.26: Freezing for correctness

```
fn validate_and_seal(data: any) {
    // ... validation logic ...
    return freeze(data)
}

fix user_record = validate_and_seal(raw_input)
// Any code that accidentally tries to mutate user_record
// will get an immediate phase error instead of silent corruption
```

11.8 Phase Errors and What They Tell You

When you violate the phase system's rules, Lattice produces clear error messages. Understanding these errors is essential for working effectively with phases.

11.8.1 Mutating a Crystal Value

The most common phase error: trying to modify something that is frozen.

Listing 11.27: Crystal mutation error

```
fix names = freeze(["Alice", "Bob"])
// names.push("Charlie")
// Error: cannot mutate crystal value
```

This error means you tried to call a mutating method (`push`, `set`, `pop`, etc.) on a crystal value. The fix is to either `thaw()` the value first or restructure your code so the mutation happens before the freeze.

11.8.2 Assigning to a Crystal Binding

Listing 11.28: Crystal assignment error

```
fix count = 42
// count = 43
// Error: cannot reassign fix binding 'count'
```

A `fix` binding cannot be reassigned. If you need a new value, create a new binding:

Listing 11.29: Working with crystal bindings

```
fix count = 42
fix new_count = count + 1
print(new_count)  // 43
```

11.8.3 Strict Mode: `let` Not Allowed

Listing 11.30: Strict mode rejects `let`

```
#mode strict

// let x = 10
// Error: strict mode: use 'flux' or 'fix' instead of 'let'
//        for binding 'x'

flux x = 10  // Use this instead
```

In strict mode, every binding must have an explicit phase. This forces you to think about mutability at every declaration point.

11.8.4 Strict Mode: Phase Mismatch

Listing 11.31: Strict mode phase mismatch

```
#mode strict

fix frozen_val = freeze(42)
// flux reassigned = frozen_val
// Error: cannot bind crystal value with flux for 'reassigned'
```

In strict mode, the phase checker tracks the phase of every expression and ensures that bindings are consistent. Binding a crystal expression with `flux` is an error because it would create a contradiction: the value is crystal but the binding says it should be fluid.

11.8.5 Strict Mode: Freezing an Already Crystal Value

Listing 11.32: Double-freeze error in strict mode

```
#mode strict

fix data = freeze([1, 2, 3])
// fix again = freeze(data)
// Error: strict mode: cannot freeze an already crystal value
```

This error helps catch redundant operations. If a value is already crystal, freezing it again is pointless—and in strict mode, Lattice tells you so.

11.8.6 Strict Mode: Thawing an Already Fluid Value

Listing 11.33: Thawing a fluid value in strict mode

```
#mode strict

flux data = [1, 2, 3]
// flux copy = thaw(data)
// Error: strict mode: cannot thaw an already fluid value
```

Similarly, thawing a fluid value is redundant. If you want an independent copy of a fluid value, use `clone()` instead.

11.8.7 Strict Mode: Fluid Values in spawn

Listing 11.34: Fluid value crossing spawn boundary

```
#mode strict

flux shared_counter = 0

scope {
    spawn {
        // print(shared_counter)
        // Error: strict mode: cannot use fluid binding
        //        'shared_counter' across thread boundary in spawn
    }
}
```

The phase checker in `src/phase_check.c` includes a special `pc_check_spawn_expr()` function that walks expressions inside `spawn` blocks and errors if any fluid binding is referenced. This is one of the most valuable strict mode checks: it prevents data races at compile time.

11.8.8 Annotation Violations

Listing 11.35: Phase annotation violation

```
#mode strict

@crystal
flux temperature = 72.5
// Error: @crystal annotation violated: initializer for
//        'temperature' is fluid
```

Phase annotations (`@fluid`, `@crystal`) are assertions about what phase a value should have. If the initializer's phase contradicts the annotation, the phase checker produces an error. We cover annotations in detail in Chapter 14.

Reading Phase Errors

Phase errors always tell you three things:

1. **What went wrong**: the specific phase violation.
2. **Where it happened**: the variable name or expression involved.
3. **Why it matters**: strict mode errors explain the rule being enforced.

When you see a phase error, ask yourself: "Should this value be mutable here, or should I have frozen it earlier?" The answer usually reveals the right fix.

11.9 Querying Phase at Runtime

The built-in `phase_of()` function returns the current phase of any value as a string:

Listing 11.36: Using `phase_of()` for runtime inspection

```
flux counter = 0
fix pi = 3.14159
let name = "Lattice"

print(phase_of(counter))  // "fluid"
print(phase_of(pi))       // "crystal"
print(phase_of(name))     // "unphased"
```

The four possible return values are:

- `"fluid"` — the value is mutable (`VTAG_FLUID`).
- `"crystal"` — the value is immutable (`VTAG_CRYSTAL`).
- `"unphased"` — the value has no explicit phase (`VTAG_UNPHASED`).
- `"sublimated"` — the value is permanently immutable (`VTAG_SUBLIMATED`).

The `phase_of()` function is implemented as a native built-in registered in `src/runtime.c`. It inspects the `phase` field of the `LatValue` struct directly, making it a zero-cost query.

`phase_of()` is particularly useful in library code that needs to handle values of any phase:

Listing 11.37: Phase-aware library function

```
fn safe_append(collection: any, item: any) {
    if phase_of(collection) == "crystal" {
        // Can't modify crystal, return a new thawed copy
        flux copy = thaw(collection)
        copy.push(item)
        return freeze(copy)
    }
    // Fluid: modify in place
    collection.push(item)
    return collection
}

flux items = [1, 2, 3]
safe_append(items, 4)
print(items)  // [1, 2, 3, 4]

fix frozen = freeze([10, 20])
fix updated = safe_append(frozen, 30)
print(updated)  // [10, 20, 30]
```

11.10 Summary: The Phase Lifecycle

Let us put it all together. A value's phase lifecycle in Lattice looks like this:

1. A value is **created** via a constructor, literal, or expression. It begins as `VTAG_UNPHASED`.
2. When bound with `flux`, it transitions to `VTAG_FLUID` and lives on the FluidHeap under GC management.
3. When bound with `fix` or passed through `freeze()`, it transitions to `VTAG_CRYSTAL` and may be relocated to a CrystalRegion.
4. `thaw()` creates a *new* fluid copy from a crystal value. `clone()` creates a copy at the same phase.
5. `sublimate()` converts to the `VTAG_SUBLIMATED` terminal state—no going back.
6. `forge`, `anneal`, `crystallize`, and `borrow` provide scoped and structured phase transitions.

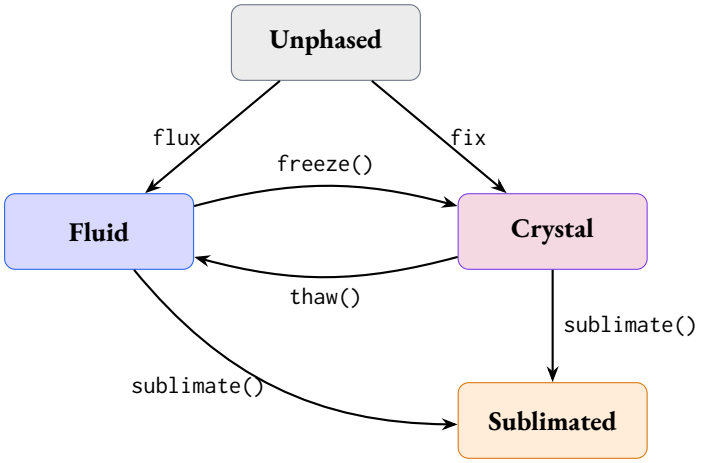


Figure 11.1: The phase transition diagram for Lattice values.

11.11 Exercises

1. **Phase Explorer.** Write a function `describe_phase(val: any)` that takes any value and prints a message describing its phase. Include different messages for fluid, crystal, unphased, and sublimated values. Test it with values of each phase.

2. **Immutable Stack.** Implement a stack data structure using frozen arrays. Your `push` operation should return a *new* frozen array with the element added, and your `pop` operation should return a tuple of the top element and the remaining frozen array. The original stack should never change.

3. **Configuration Builder.** Write a `build_config()` function that uses a `forge` block to construct a configuration map. The function should accept an array of key-value tuples and build the frozen map from them.

4. **Phase-Safe Merge.** Write a function `merge_arrays(a: any, b: any)` that concatenates two arrays regardless of their phases. If either input is crystal, the function should thaw it before merging. The result should always be frozen.

5. **Error Collector.** In strict mode, write a program that deliberately triggers each of the phase errors described in Section 11.8 (one at a time). Comment out each error after confirming the message, and write a brief comment explaining what the error means.

What's Next. Now that we understand the phase system's semantics, we are ready to peer beneath the surface and ask: *where do these values actually live?* In Chapter 12, we explore the dual-heap memory architecture that makes the phase system possible—the FluidHeap for mutable values,

the CrystalRegion for frozen data, and the ephemeral BumpArena for short-lived temporaries. Understanding this architecture will give you the intuition to write code that is not only correct but efficient.

Chapter 12

Memory and Arenas

In the previous chapter we learned *what* phases mean for your code. Now we ask: *where does the data actually go?* When you freeze a value, what happens to the bytes? When the garbage collector runs, how does it know to skip crystal values? Why does freezing sometimes make your program *faster*?

Lattice uses a *dual-heap architecture* that physically separates mutable and immutable data into different regions of memory. This is not a coincidence or an optimization afterthought—it is a direct consequence of the phase system's design. Fluid values live in one world, crystal values in another, and the boundary between them is enforced all the way down to the allocator.

12.1 The FluidHeap: Mark-Sweep GC for Mutable Values

Every mutable value in Lattice lives on the *FluidHeap*—a garbage-collected heap that uses a mark-sweep algorithm to reclaim memory.

When you write a `flux` declaration, the value's internal data (strings, array buffers, map entries) is allocated through the FluidHeap's `fluid_alloc()` function. You can see the implementation in `src/memory.c`:

Listing 12.1: Fluid values live on the GC-managed heap

```
flux names = ["Alice", "Bob", "Charlie"]
// The array buffer and each string are allocated via fluid_alloc()
// The FluidHeap tracks every allocation in a linked list
```

12.1.1 How the FluidHeap Works

The `FluidHeap` structure, defined in `include/memory.h`, maintains a linked list of every allocation it has made:

- Each allocation is wrapped in a `FluidAlloc` node containing the pointer, its size, a `marked` flag, and a link to the next allocation.
- The heap tracks `total_bytes`, `alloc_count`, and `gc_threshold` (initially 1 MB).
- When the total bytes exceed the GC threshold, a collection cycle is triggered.

The allocation path is straightforward: `fluid_alloc()` calls `malloc()`, wraps the result in a `FluidAlloc` header, prepends it to the linked list, and updates the byte counters. Deallocation via `fluid_dealloc()` walks the list, finds the matching pointer, unlinks the node, and frees the memory.

12.1.2 Mark-Sweep Collection

The GC uses a classic two-phase algorithm:

1. **Mark phase.** Starting from the *roots* (the VM stack, global environment, struct metadata, open upvalues, call frame upvalues, and the module cache), the collector recursively traverses every reachable value and marks its heap allocations. The `gc_mark_value()` function in `src/gc.c` handles this traversal.
2. **Sweep phase.** The collector walks the entire `FluidHeap` allocation list. Any allocation not marked in step 1 is unreachable—it gets freed. Marked allocations have their mark bit cleared for the next cycle.

Listing 12.2: GC reclaims unreachable fluid values

```
fn create_temporary() {
    flux temp = [1, 2, 3, 4, 5]
    // temp is allocated on the FluidHeap
    return temp.len()
    // After return, temp is unreachable
    // The next GC cycle will free its memory
}

flux length = create_temporary()
print(length)  // 5
```

Crystal Values Are Invisible to the GC

A critical optimization: `gc_mark_value()` in `src/gc.c` checks the `region_id` field of every value. If the region ID is anything other than `REGION_NONE`—meaning the value is arena-backed, ephemeral, interned, or a constant—the marker returns immediately. Crystal values stored in a CrystalRegion are *never traversed* during the mark phase. This means the GC's pause time scales with the number of *fluid* values, not the total number of values. The more data you freeze, the less work the GC has to do.

12.1.3 The GC Threshold and Adaptive Growth

The garbage collector does not run on every allocation. Instead, it triggers when the object count exceeds the `next_gc` threshold. After each collection, the threshold adapts:

$$\text{next\_gc} = \text{surviving\_objects} \times 2$$

This growth factor (defined as `GC_GROWTH_FACTOR` in `src/gc.c`) ensures that the GC runs more frequently when there are many objects and less frequently when most objects survive. The minimum threshold is 256 objects (`GC_INITIAL_THRESHOLD`), preventing thrashing on small programs.

Listing 12.3: Adaptive GC in action

```
flux items = []
for i in 0..10000 {
    items.push(to_string(i))
    // The GC may trigger periodically as items grows
    // Each cycle, the threshold adjusts based on survivors
}
print(items.len())  // 10000
```

12.2 The CrystalRegion: Arena-Backed Storage for Immutable Values

When you freeze a value, its data is deep-cloned into a *CrystalRegion*—an arena-based allocator that provides two key benefits: cache locality and O(1) bulk deallocation.

12.2.1 What Is an Arena?

An arena (also called a region allocator or bump allocator) is a memory management strategy where you allocate objects by bumping a pointer forward within a pre-allocated block of memory called a *page*. When you are done with the arena, you free all its memory at once by freeing the pages. Individual objects are never freed independently.

Arena Allocation

An *arena* allocates memory by advancing a pointer within contiguous pages. Allocation is O(1)—just bump the pointer. Deallocation is also O(1)—free all pages at once. The trade-off: you cannot free individual objects; you free the entire arena or nothing.

This is a perfect fit for crystal values. Once frozen, they are immutable and their lifetimes are tied together: when the last reference to a frozen value is dropped, the entire region can be reclaimed.

12.2.2 Region Structure

The `CrystalRegion` struct, defined in `include/memory.h`, contains:

- An `id` (a monotonically increasing `RegionId`) that uniquely identifies this region.
- An `epoch` timestamp for generational tracking.
- A linked list of `ArenaPage` structures, each backed by a 4 KB data buffer (`ARENA_PAGE_SIZE`).
- A running total of bytes used across all pages.

Allocation within a region is handled by `arena_alloc()` in `src/memory.c`. It uses 8-byte alignment and tries to fit the request in the current head page. If the page is full, a new page is allocated and prepended to the list. Oversized requests get a dedicated page.

Listing 12.4: Freeze allocates into a CrystalRegion

```
flux large_dataset = []
for i in 0..1000 {
    large_dataset.push(i * i)
}

// freeze() clones the data into a new CrystalRegion
fix snapshot = freeze(large_dataset)

// The 1000 integers and the array buffer are now packed
// into contiguous arena pages---great for cache performance
```

12.2.3 How Freeze Moves Data

When `freeze()` is called, the runtime performs these steps:

1. A new `CrystalRegion` is created via `region_create()` on the `RegionManager`.
2. The global arena pointer (`g_arena` in `src/value.c`) is set to this new region.
3. `value_deep_clone()` is called. Because `g_arena` is now set, every internal allocation call (`lat_alloc`, `lat_strdup`, etc.) is routed to `arena_alloc()` instead of the normal heap or FluidHeap.
4. The phase tag is set to `VTAG_CRYSTAL` recursively.
5. The arena pointer is cleared.
6. Each cloned value's `region_id` is set to the region's ID, marking it as arena-backed.

This is the key mechanism: the deep clone into an arena creates a completely independent copy whose memory is managed by the region, not by the FluidHeap or the GC. The allocation routing logic in `src/value.c` makes this transparent:

- `lat_alloc()` checks `g_arena` first. If set, it calls `arena_alloc()`. Otherwise, it tries the FluidHeap, and as a last resort, raw `malloc()`.
- `lat_free()` checks `g_arena`—if set, it is a no-op (arena memory is freed in bulk).

Region IDs and GC Coordination

Every value cloned into an arena carries the region's ID in its `region_id` field. During GC, the marker uses this field to skip arena-backed values. During region collection, the region manager identifies which regions are still reachable (their IDs appear in the set of live region IDs) and frees the rest. This coordination is implemented in `region_collect()` in `src/memory.c`, which uses a sorted binary search for efficient reachability checks.

12.2.4 Region Collection

Crystal regions are freed by the `region_collect()` function on the `RegionManager`. During a GC cycle, the runtime builds a set of *reachable region IDs* by scanning all live values' `region_id` fields. Any region whose ID does not appear in this set is unreachable—its pages are freed in bulk.

This is the beauty of arena allocation: freeing a CrystalRegion is a simple walk through its page list, freeing each page. No per-object destructor calls, no pointer chasing through a complex object graph. For a region containing thousands of frozen values, the cost is the same as freeing a handful of pages.

Listing 12.5: Regions freed when no longer reachable

```
fn make_snapshot() {
    flux data = [1, 2, 3, 4, 5]
    return freeze(data)
}

flux snapshot = make_snapshot()
print(snapshot)  // [1, 2, 3, 4, 5]

// If snapshot goes out of scope or is reassigned,
// the CrystalRegion backing it becomes unreachable
// and is freed during the next GC cycle
snapshot = nil  // The region can now be collected
```

12.2.5 The RegionManager

The `RegionManager` in `include/memory.h` is the bookkeeper for all crystal regions. It maintains:

- A dynamic array of `CrystalRegion` pointers.
- A monotonically increasing `next_id` counter for assigning unique region IDs.
- An `epoch` counter for generational tracking.
- Statistics: total allocations, peak region count, cumulative data bytes.

New regions are created with `region_create()`, which allocates a fresh region with a single initial page and assigns it the next available ID. The manager grows its array dynamically as regions accumulate.

12.3 The Ephemeral BumpArena: Short-Lived Temporaries

Not every allocation deserves the overhead of GC tracking or the permanence of an arena region. For short-lived temporary values—intermediate computation results, format strings, temporary buffers—Lattice provides the `BumpArena`.

The BumpArena is a lightweight bump allocator that can be *reset* without freeing its pages. This means the same memory pages are reused across multiple reset cycles, avoiding the cost of repeated page allocation.

12.3.1 How the BumpArena Works

The `BumpArena` struct in `include/memory.h` maintains:

- A `pages` pointer to the current page.
- A `first_page` pointer to the head of the page chain (preserved across resets).
- A running total of bytes allocated since the last reset.

Allocation via `bump_alloc()` (in `src/memory.c`) works in three steps:

1. Try the current page. If there is room, bump the `used` pointer and return.
2. Try the next page in the chain (left over from a previous cycle). If there is room, advance to that page.
3. Allocate a new page. Oversized requests get a dedicated page larger than the default 4 KB.

All allocations are 8-byte aligned via `(size + 7) & ~7`, ensuring proper alignment for any data type.

12.3.2 The Reset Cycle

The key feature of the BumpArena is `bump_arena_reset()`. Instead of freeing pages, it simply sets each page's `used` counter back to zero and resets the current page pointer to the first page. The page chain remains intact, so subsequent allocations reuse existing memory without any system calls.

Listing 12.6: Ephemeral allocations for temporaries

```
// Inside the VM, each expression evaluation may allocate
// temporary strings for interpolation, display, or hashing.
// These temporaries live in the BumpArena and are cleared
// at safe points (e.g., between statements).

flux message = "Count: ${to_string(counter)}"
// The interpolation creates temporary strings in the BumpArena
// They are freed in bulk at the next safe point
```

Values allocated in the BumpArena have their `region_id` set to `REGION_EPHEMERAL` (defined as `(size_t)-2` in `include/value.h`). This sentinel value tells both the GC marker and the value destructor to skip these allocations—they are managed by the arena lifecycle, not by normal heap operations.

Performance Benefit

The BumpArena is one of the reasons Lattice can handle string-heavy operations efficiently. String interpolation, formatting, and display operations generate many short-lived strings. Rather than allocating and freeing each one individually (which would pressure the GC), they are bump-allocated and cleared in bulk. This reduces GC pressure and improves throughput.

12.4 How Freeze Moves a Value from Heap to Arena

Let us walk through the complete memory journey of a value as it transitions from fluid to crystal. Understanding this flow gives you an intuition for the cost and benefit of freezing.

12.4.1 Step by Step

Consider this code:

Listing 12.7: A freeze in slow motion

```
flux temperatures = [20.1, 21.5, 19.8]
fix snapshot = freeze(temperatures)
```

Here is what happens in memory:

1. **Initial state.** `temperatures` is a fluid array. Its element buffer (`elems`) is allocated on the FluidHeap. The three float values are stored inline in the buffer. The `phase` tag is `VTAG_FLUID`, and `region_id` is `REGION_NONE`.

2. **Region creation.** `freeze()` calls `region_create()` on the RegionManager. A new CrystalRegion is born with a fresh ID and a single 4 KB page.

3. **Arena activation.** The thread-local `g_arena` pointer is set to the new region. From this point, all `lat_alloc()` calls are routed to `arena_alloc()`.

4. **Deep clone.** `value_deep_clone()` copies the array. The new element buffer is allocated in the arena (not the FluidHeap). Each float value is copied into the new buffer. For compound elements, the clone recurses.

5. **Phase tagging.** `set_phase_recursive()` walks the cloned value tree and sets every `phase` tag to `VTAG_CRYSTAL`.

6. **Region ID assignment.** The cloned value's `region_id` is set to the region's ID.

7. **Arena deactivation.** `g_arena` is set back to `NULL`. Subsequent allocations go through the normal FluidHeap path.

8. **Binding.** The crystal value is stored in the `fix` binding `snapshot`.

After this process, `temperatures` still lives on the FluidHeap (and can still be modified), while `snapshot` lives in its own CrystalRegion with all data packed into contiguous arena pages.

Property	FluidHeap	CrystalRegion
Allocation style	Individual `malloc()`	Arena bump allocation
Deallocation	Per-object via GC sweep	Bulk page free
Cache locality	Scattered	Contiguous
GC interaction	Mark & sweep	Skipped during marking
Mutability	Mutable	Immutable
`region_id`	`REGION_NONE`	Region-specific ID

Table 12.1: Comparing FluidHeap and CrystalRegion memory characteristics.

12.4.2 Memory Layout Comparison

12.5 String Interning and Why It Matters

Lattice interns certain strings—struct field names, map keys used internally, and explicitly interned strings—into a global `InternTable`. Interning means that each unique string exists at most once in memory, and all references to that string point to the same canonical copy.

12.5.1 How Interning Works

The intern table, implemented in `src/intern.c`, is a hash table using open addressing with linear probing:

- The hash function is DJB2 (`h = h * 33 + c`), a fast string hash.
- The table starts at 256 entries and grows (doubles) when the load factor exceeds 50%.
- `intern()` looks up the string. If found, it returns the existing pointer. If not, it `strdup()`s the string into the table and returns the new pointer.

The result: two interned strings with the same content are guaranteed to have the same pointer. This means pointer comparison (==) replaces `strcmp()` for interned strings, turning an O(n) comparison into an O(1) operation.

12.5.2 What Gets Interned

In the Lattice runtime, the following strings are automatically interned:

- **Struct field names.** When a struct is created (via `value_struct()` in `src/value.c`), each field name is passed through `intern()`. This is visible in the constructor: `val.as.strct.field_names[i] = (char *)intern(field_names[i])`.

- **Explicitly interned strings.** The `value_string_interned()` constructor creates a string value whose `region_id` is set to `REGION_INTERNED`.

Interned strings are never freed during normal program execution (they are owned by the global intern table) and are cleaned up only when `intern_free()` is called at program exit.

12.5.3 Why Interning Matters for Phase

String interning interacts with the phase system in several important ways:

1. **GC safety.** Interned strings have `region_id = REGION_INTERNED`. The GC marker skips them (they are not on the FluidHeap), and the value destructor skips them (they are owned by the intern table). This prevents double-frees and dangling pointers.
2. **Efficient deep cloning.** When `value_deep_clone()` encounters an interned string, it does *not* copy the string data. Instead, it reuses the same pointer and copies the `REGION_INTERNED` marker. This makes cloning structs (which have interned field names) cheaper.
3. **Fast field lookup.** Because field names are interned, struct field access can use pointer comparison instead of string comparison, speeding up field lookups in hot paths.

Listing 12.8: Interning makes struct operations faster

```
struct Point {
    x: Int,
    y: Int
}

// The field names "x" and "y" are interned once.
// Every Point instance shares the same name pointers.
// Field lookup compares pointers, not strings.

flux p1 = Point { x: 10, y: 20 }
flux p2 = Point { x: 30, y: 40 }
// p1.as.strct.field_names[0] == p2.as.strct.field_names[0]
// (same pointer, thanks to interning)
```

12.6 The DualHeap: Putting It All Together

The `DualHeap` struct in `include/memory.h` ties the FluidHeap and RegionManager together:

- `fluid` — a pointer to the `FluidHeap` for GC-managed mutable allocations.

- `regions` — a pointer to the `RegionManager` for arena-backed crystal storage.

At VM startup, `dual_heap_new()` creates both subsystems. At shutdown, `dual_heap_free()` tears them down, freeing all remaining fluid allocations and all crystal regions.

The DualHeap is set as the active heap via `value_set_heap()` at the start of program execution. This call installs the `DualHeap` into a thread-local global, so all subsequent `lat_alloc()` calls are routed through it.

Listing 12.9: The DualHeap manages both worlds

```
// At program startup:
// DualHeap is created
//   -> FluidHeap for flux values
//   -> RegionManager for fix values

flux mutable_data = [1, 2, 3]     // allocated via FluidHeap
fix frozen_data = freeze([4, 5])  // cloned into CrystalRegion

// At program shutdown:
// DualHeap frees everything
//   -> FluidHeap frees all remaining fluid allocations
//   -> RegionManager frees all crystal regions and their pages
```

12.7 GC Flags: Tuning the Collector

Lattice provides command-line flags to control garbage collection behavior. These are useful for debugging memory issues, benchmarking, and tuning performance.

12.7.1 –gc: Enable the Garbage Collector

By default, the GC is enabled in the bytecode VM. The `--gc` flag explicitly enables garbage collection:

Listing 12.10: Running with GC enabled

```
// Command line:
// lattice --gc my_program.lat
```

The GC state is managed by the `GC` struct in `include/gc.h`, which tracks:

- `all_objects`: linked list of all GC-tracked allocations.

- `object_count`: current number of tracked objects.
- `next_gc`: threshold for the next collection.
- `enabled`: whether the GC is active.
- `total_collected` and `total_cycles`: lifetime statistics.

12.7.2 `--gc-stress`: Collect on Every Allocation

The `--gc-stress` flag enables *stress mode*, where the collector runs on every single allocation. This is invaluable for catching GC-related bugs: use-after-free, missing root registrations, and incorrect mark implementations.

Listing 12.11: Stress testing the GC

```
// Command line:
// lattice --gc-stress my_program.lat

// Every allocation triggers a full mark-sweep cycle.
// Extremely slow, but catches subtle memory bugs.
```

In stress mode, `gc_maybe_collect()` in `src/gc.c` always triggers a collection:

```
if (gc->stress || gc->object_count >= gc->next_gc) { gc_collect(gc, vm_ptr);
}
```

Do Not Use Stress Mode in Production

GC stress mode runs a full mark-sweep cycle on every allocation, making it orders of magnitude slower than normal execution. Use it only for debugging and testing.

12.7.3 `--gc-incremental`: Spread Work Across Safe Points

The `--gc-incremental` flag enables *incremental collection*, which spreads GC work across multiple safe points to reduce pause times.

Instead of stopping the world for a full mark-sweep, the incremental collector operates as a state machine with four phases, defined as `GCPhase` in `include/gc.h`:

1. **Idle** (`GC_PHASE_IDLE`). No collection in progress. The collector checks if the threshold is exceeded and, if so, transitions to marking.

2. **Mark roots** (`GC_PHASE_MARK_ROOTS`). Scans all VM roots (stack, globals, upvalues, module cache) and pushes compound values onto a *gray worklist*. Leaf types (strings, buffers) are marked directly without being pushed.

3. **Mark trace** (`GC_PHASE_MARK_TRACE`). Pops values from the gray worklist and traces their children, up to a configurable `mark_budget` per step (default: 64 values). When the worklist is empty, the roots are re-scanned once (to catch mutations during marking), then the sweep begins.

4. **Sweep** (`GC_PHASE_SWEEP`). Walks the object list, freeing unmarked objects, up to a `sweep_budget` per step (default: 128 objects). When the sweep completes, the cycle is done and the threshold is updated.

Listing 12.12: Incremental GC for lower latency

```
// Command line:
// lattice --gc-incremental my_program.lat

// The GC spreads its work across VM safe points
// (typically at OP_RESET_EPHEMERAL boundaries),
// keeping individual pauses short.
```

No Write Barriers

Lattice's incremental collector does not use write barriers. Instead, it compensates by re-scanning roots after the initial mark trace completes (the `roots_rescanned` flag in the `GC` struct). This conservative approach is correct but may do slightly more work than a write-barrier-based collector. New objects allocated during an incremental cycle are born "black" (marked), so the sweep phase will not prematurely free them.

12.7.4 GC Statistics

The `GC` struct tracks several statistics that are useful for performance analysis:

Similarly, the RegionManager tracks:

12.8 Special Region IDs

Every `LatValue` has a `region_id` field that indicates where its data lives. The runtime defines several sentinel values (in `include/value.h`) that identify special allocation regions:

Field	Meaning
`object_count`	Current number of GC-tracked objects
`bytes_allocated`	Total bytes currently allocated under GC
`total_collected`	Lifetime count of objects freed by the GC
`total_cycles`	Number of GC cycles completed
`next_gc`	Object count threshold for next collection

Table 12.2: GC statistics fields.

Field	Meaning
`count`	Number of live crystal regions
`total_allocs`	Lifetime count of regions created
`peak_count`	Maximum simultaneous live regions
`cumulative_data_bytes`	Total bytes allocated across all regions

Table 12.3: RegionManager statistics fields.

Any `region_id` that is not one of these sentinels is a real CrystalRegion ID. The GC marker, the value destructor, and the deep-clone function all check `region_id` to decide how to handle a value.

12.9 Memory Best Practices

Armed with knowledge of the dual-heap architecture, here are practical guidelines for memory-efficient Lattice programs:

Constant	Value	Meaning
`REGION_NONE`	`(size_t)-1`	Normal heap (FluidHeap or `malloc`)
`REGION_EPHEMERAL`	`(size_t)-2`	BumpArena (temporary, reset-managed)
`REGION_INTERNED`	`(size_t)-3`	Intern table (never freed individually)
`REGION_CONST`	`(size_t)-4`	Constant pool (borrowed, not freed)

Table 12.4: Special region ID sentinel values.

Memory Guidelines

- **Freeze early, freeze often.** Crystal values reduce GC pressure because the marker skips them entirely. If data will not change after initialization, freeze it.
- **Use forge blocks for complex construction.** Building a value in a forge block localizes the mutable temporaries and produces a single crystal result in one arena, maximizing cache locality.
- **Avoid unnecessary thaw-modify-freeze cycles.** Each cycle involves a deep clone. If you need to make multiple modifications, thaw once, apply all changes, and freeze once.
- **Let large fluid values die quickly.** The GC threshold adapts based on surviving objects. If large temporary fluid values are kept alive unnecessarily, the GC threshold grows and memory usage stays high. Drop references as soon as you are done.
- **Use incremental GC for latency-sensitive applications.** If your program handles real-time events (web servers, game loops), `--gc-incremental` keeps pauses bounded.

12.10 Exercises

1. **Freeze Cost.** Write a program that creates a deeply nested array (arrays of arrays of arrays, at least 3 levels deep with 100 elements at each level). Time how long it takes to freeze the entire structure. Then time how long it takes to access elements of the frozen copy versus the original fluid copy. What do you observe?

2. **Snapshot Chain.** Write a function that maintains a "history" of an evolving value by freezing a snapshot after every modification. Store the snapshots in an array. How many distinct CrystalRegions do you think are created?

3. **GC Pressure Experiment.** Write a program that creates many short-lived fluid values in a loop (e.g., build and discard 10,000 arrays of 100 elements each). Run it with `--gc` and observe the behavior. Then modify the program to freeze values that persist. Reason about how the GC threshold changes in each case.

4. **Intern Detective.** Create two structs of the same type with different field values. Use `phase_of()` and `type_of()` to inspect them. Reason about which parts of the structs share memory (interned field names) and which have independent allocations (field values).

What's Next. We have seen how individual values transition between fluid and crystal, and where they live in memory. But what about data structures where *some fields* should be mutable and *others* should be frozen? In Chapter 13, we explore alloy structs—structures with per-field phase declarations—and the reactive bond system that lets values respond automatically when their neighbors change phase.

Chapter 13

Alloys and Reactive Bonds

So far, the phase system has been all-or-nothing: a value is either entirely fluid or entirely crystal. Freeze an array, and every element freezes. Freeze a struct, and every field hardens. But real-world data is rarely so uniform.

Consider a user session object. The user's ID and email should be immutable once established—nobody should accidentally overwrite them. But the session's last-active timestamp needs to update on every request, and the permissions cache might need to refresh periodically. You need a struct where *some fields are crystal and others are fluid*—a material with mixed properties.

In materials science, such a material is called an *alloy*: a combination of elements with different characteristics fused into a single structure. Lattice borrows this metaphor. An alloy struct has per-field phase declarations, so each field can have its own mutability. And once you have mixed-phase data, Lattice goes further with *reactive bonds*—automatic relationships between variables that propagate phase changes—and *seeds*—validation contracts that guard the crystallization process.

13.1 Alloy Structs: Per-Field Phase Declarations

An alloy struct declares the phase of each field individually using `flux` and `fix` annotations in the struct definition:

Listing 13.1: Defining an alloy struct

```
struct Session {
    fix user_id: String,
    fix email: String,
    flux last_active: String,
    flux permissions: Array
}
```

When you create an instance of this struct, the runtime applies per-field phase enforcement. The `fix` fields are automatically frozen, while the `flux` fields remain fluid:

Listing 13.2: Creating an alloy struct instance

```
flux session = Session {
    user_id: "usr_42",
    email: "alice@example.com",
    last_active: "2024-01-15T10:30:00Z",
    permissions: ["read", "write"]
}

// Crystal fields are locked
print(session.user_id)     // "usr_42"
// session.user_id = "usr_99"  // Error: cannot assign to frozen field 'user_id'

// Fluid fields can change
session.last_active = "2024-01-15T10:31:00Z"
session.permissions.push("admin")
print(session.last_active)  // "2024-01-15T10:31:00Z"
```

Alloy

An *alloy* is a struct (or map) with per-field phase declarations. Individual fields carry their own phase tag, independent of the overall struct's phase. Crystal fields reject mutation; fluid fields allow it. The struct itself can be fluid (allowing field reassignment for fluid fields) while its crystal fields remain permanently locked.

13.1.1 How Alloys Work Under the Hood

The per-field phase system is implemented through the `field_phases` array in the `LatValue` struct (see `include/value.h`). When a struct has per-field phases, `field_phases` is a dynamically allocated array of `PhaseTag` values—one for each field. If `field_phases` is `NULL`, all fields inherit the struct's overall phase.

During struct construction in the VM (`src/stackvm.c`), the runtime checks for a per-field phase declaration stored in the global environment under the key `__struct_phases_<StructName>`. This is an array of integer phase codes:

- `0` means `PHASE_FLUID` — the field stays mutable.
- `1` means `PHASE_CRYSTAL` — the field is automatically frozen at construction.
- Any other value means the field inherits the struct's overall phase.

When field assignment is attempted (`OP_SET_FIELD`), the VM checks the `field_phases` array:

Listing 13.3: Field-level phase enforcement

```
// Internally, the VM checks:
// if (obj.as.strct.field_phases[i] == VTAG_CRYSTAL)
//     ERROR: "cannot assign to frozen field 'user_id'"
```

This enforcement happens at the deepest level of the runtime—there is no way to bypass it through clever coding. The field's phase tag is a property of the value itself, checked on every write.

13.1.2 Partial Freeze with `except`

You can also create alloy-like behavior dynamically by freezing a struct with exceptions:

Listing 13.4: Freezing a struct with exceptions

```
struct Config {
    host: String,
    port: Int,
    debug: Bool,
    log_level: String
}

flux config = Config {
    host: "localhost",
    port: 8080,
    debug: true,
    log_level: "info"
}

// Freeze everything except debug and log_level
freeze(config, except: ["debug", "log_level"])

// config.host = "evil.com"   // Error: cannot assign to frozen field 'host'
// config.port = 9999         // Error: cannot assign to frozen field 'port'
config.debug = false           // OK: debug was exempted
config.log_level = "debug"     // OK: log_level was exempted
```

The `except` clause is compiled to the `OP_FREEZE_EXCEPT` opcode. Under the hood (in `src/stackvm.c`), this opcode:

1. Allocates the `field_phases` array if it does not already exist.
2. Iterates through each field, freezing it unless it appears in the exception list.
3. Sets the exempted fields' phases to `VTAG_FLUID`.

The same mechanism works for maps—each key can have its own phase via the `key_phases` map stored alongside the map's data.

13.2 Designing Data with Mixed Mutability

Alloy structs are a powerful design tool. Here are patterns that emerge naturally from per-field phases.

13.2.1 Identity + State Pattern

Separate immutable identity from mutable state:

Listing 13.5: Identity + State pattern

```
struct Player {
    fix id: Int,
    fix name: String,
    flux health: Int,
    flux position_x: Float,
    flux position_y: Float,
    flux inventory: Array
}

flux hero = Player {
    id: 1,
    name: "Aria",
    health: 100,
    position_x: 0.0,
    position_y: 0.0,
    inventory: ["sword", "shield"]
}

// Identity is locked forever
print(hero.name)  // "Aria"

// State changes freely
hero.health = hero.health - 15
hero.position_x = 3.5
hero.inventory.push("potion")
```

13.2.2 Configuration + Runtime Pattern

Lock configuration values while allowing runtime counters and caches to change:

Listing 13.6: Configuration + Runtime pattern

```
struct Server {
    fix host: String,
    fix port: Int,
    fix max_connections: Int,
    flux active_connections: Int,
    flux request_count: Int
}

flux server = Server {
    host: "0.0.0.0",
    port: 443,
    max_connections: 10000,
    active_connections: 0,
    request_count: 0
}

// Configuration is safe from accidental changes
// Runtime counters update freely
server.active_connections = server.active_connections + 1
server.request_count = server.request_count + 1
```

13.2.3 Append-Only Pattern

Use a crystal array field for historical records (by replacing it with a new frozen array), while keeping a mutable accumulator:

Listing 13.7: Append-only log pattern

```
struct AuditLog {
    fix entries: Array,
    flux pending: Array
}

fn add_entry(log: any, entry: any) {
    log.pending.push(entry)
    return log
}

fn flush_log(log: any) {
    flux all = thaw(log.entries)
    for item in log.pending {
        all.push(item)
    }
    // Return new log with frozen entries and empty pending
    return AuditLog {
        entries: freeze(all),
        pending: []
    }
}
```

Alloy Design Principle

When designing an alloy struct, ask: "Which fields define *what this thing is* (identity), and which fields describe *how it is right now* (state)?" Identity fields should be `fix`. State fields should be `flux`.

13.3 Reactive Bonds: `bond()`, `unbond()`, `react()`, `unreact()`

Lattice provides a reactive system that lets variables respond automatically when the phase of another variable changes. The core primitives are:

- `react(variable, callback)` — register a callback that fires when a variable's phase changes.
- `unreact(variable)` — remove all reaction callbacks for a variable.
- `bond(target, dependency, strategy)` — create a dependency relationship between variables.
- `unbond(target, dependency)` — remove a specific dependency.

13.3.1 Reactions: Responding to Phase Changes

`react()` registers a callback closure that is invoked whenever a variable undergoes a phase transition (freeze, thaw, or sublimate):

Listing 13.8: Setting up a phase reaction

```
flux temperature = 72.5

react(temperature, |phase: any| {
    print("Temperature phase changed to: " + phase)
})

freeze(temperature)
// Output: Temperature phase changed to: crystal

thaw(temperature)
// Output: Temperature phase changed to: fluid
```

The callback receives a string describing the new phase: `"crystal"`, `"fluid"`, or `"sublimated"`. Multiple callbacks can be registered for the same variable—they fire in the order they were added.

Under the hood, reactions are stored in the `LatRuntime` struct (see `include/runtime.h`). Each `RTReaction` entry associates a variable name with an array of callback closures. When `OP_FREEZE_VAR` or `OP_THAW_VAR` executes in the VM, it calls `stackvm_fire_reactions()` to invoke all registered callbacks.

Listing 13.9: Multiple reactions on one variable

```
flux sensor = 98.6

react(sensor, |phase: any| {
    print("Logger: sensor is now " + phase)
})

react(sensor, |phase: any| {
    if phase == "crystal" {
        print("Alert: sensor has been frozen!")
    }
})

freeze(sensor)
// Output: Logger: sensor is now crystal
// Output: Alert: sensor has been frozen!
```

13.3.2 Removing Reactions

`unreact()` removes all reaction callbacks registered on a variable:

Listing 13.10: Removing all reactions

```
flux value = 42
react(value, |phase: any| {
    print("phase changed: " + phase)
})

unreact(value)
freeze(value)  // No output---the callback has been removed
```

13.3.3 Bonds: Phase Dependency Relationships

While reactions let you *observe* phase changes, bonds let you *propagate* them. `bond()` creates a dependency relationship: when a dependency variable freezes, the target variable also freezes automatically.

Listing 13.11: Creating a bond between variables

```
flux primary = [1, 2, 3]
flux replica = [1, 2, 3]

bond(replica, "primary", "mirror")

// When primary freezes, replica automatically freezes too
freeze(primary)
print(phase_of(primary))  // "crystal"
print(phase_of(replica))  // "crystal" (frozen via cascade)
```

The third argument to `bond()` is the *strategy* string, which determines how the freeze cascades. The `"mirror"` strategy (the default) means the target mirrors the dependency's phase transitions.

Bond Requirements

Bonds have two requirements:

1. The dependency variable must exist at the time of bonding.
2. The target variable must not already be frozen—you cannot bond a crystal variable (it is already locked).

Violating either rule produces a runtime error: "`cannot bond undefined variable`" or "`cannot bond already-frozen variable`."

13.3.4 Removing Bonds

`unbond()` removes a specific dependency from a target:

Listing 13.12: Removing a bond

```
flux source = 100
flux follower = 100

bond(follower, "source", "mirror")
unbond(follower, "source")

freeze(source)
print(phase_of(follower))  // "fluid" (no longer bonded)
```

If removing a dependency leaves a bond entry with no dependencies, the entire bond entry is cleaned up.

13.4 Bond Strategies: Mirror, Inverse, Gate

The strategy parameter in `bond()` controls how phase transitions propagate from dependency to target.

13.4.1 Mirror Strategy

The `"mirror"` strategy is the default. When the dependency freezes, the target freezes. When the dependency thaws, the target thaws. The target mirrors whatever the dependency does:

Listing 13.13: Mirror bond strategy

```
flux leader = "hello"
flux follower = "world"

bond(follower, "leader", "mirror")

freeze(leader)
print(phase_of(follower))  // "crystal" (mirrored freeze)
```

The mirror strategy is implemented in `rt_freeze_cascade()` in `src/runtime.c`. When a variable is frozen, the cascade function walks all bonds, finds targets whose dependencies include the frozen variable, and applies the appropriate phase transition.

13.4.2 Designing Custom Strategies

Bond strategies are stored as string identifiers in the `RTBond` struct's `dep_strategies` array. The runtime matches on these strings during cascade operations. While the core runtime ships with the mirror strategy, the design is extensible—additional strategies can be recognized by the cascade logic.

Here are patterns you can build with the existing bond primitives:

Inverse Pattern

Using reactions, you can implement an inverse relationship where freezing one variable thaws another:

Listing 13.14: Inverse relationship via reactions

```
flux active_config = Map::new()
active_config.set("theme", "dark")

flux backup_config = Map::new()
backup_config.set("theme", "dark")

react(active_config, |phase: any| {
    if phase == "crystal" {
        // When active freezes, make a thawed backup
        print("Active config frozen, backup is now active")
    }
})

freeze(active_config)
// Output: Active config frozen, backup is now active
```

Gate Pattern

A gate pattern uses a boolean sentinel to control whether a freeze cascades:

Listing 13.15: Gate pattern with reactions

```
flux gate_open = true
flux protected_data = [1, 2, 3]

react(protected_data, |phase: any| {
    if phase == "crystal" {
        print("Data has been sealed")
    }
})

fn conditional_freeze(data: any, gate: any) {
    if gate {
        return freeze(data)
    }
    return data
}

// With gate open, freeze proceeds
fix sealed = conditional_freeze(protected_data, gate_open)
// Output: Data has been sealed
```

13.5 Pressure: `pressurize()`, `depressurize()`, `pressure_of()`

Pressure is a constraint system for *structural mutations* on collections. While the phase system controls whether a value can be mutated at all, pressure controls *how* it can be mutated—specifically, whether a collection can grow, shrink, or both.

Listing 13.16: Applying pressure constraints

```
flux log_entries = ["boot", "init"]

// Apply pressure: no shrinking allowed
pressurize("log_entries", "no_shrink")

log_entries.push("request received")  // OK: growing is allowed
print(log_entries)  // ["boot", "init", "request received"]

// log_entries.pop()  // Error: pressure constraint violated (no_shrink)
```

13.5.1 Pressure Modes

The `pressurize()` function accepts a variable name (as a string) and a mode string. Four modes are available:

Mode	Effect
`"no_grow"`	Blocks `push()`, `insert()`, and other size-increasing operations.
`"no_shrink"`	Blocks `pop()`, `remove_at()`, and other size-reducing operations.
`"no_resize"`	Blocks both growing and shrinking—the collection's size is locked.
`"read_heavy"`	An optimization hint indicating the collection will be read much more often than written.

Table 13.1: Pressure modes and their effects.

Listing 13.17: Different pressure modes

```
flux scores = [85, 92, 78, 95]

// Lock the size entirely
pressurize("scores", "no_resize")

// scores.push(100)      // Error: pressure constraint (no_resize)
// scores.pop()           // Error: pressure constraint (no_resize)
scores[0] = 90            // OK: element modification is still allowed
print(scores)  // [90, 92, 78, 95]
```

Pressure vs. Phase

Pressure and phase are orthogonal. A fluid value under `"no_resize"` pressure can still have its *elements* modified—you just cannot add or remove elements. A crystal value rejects *all* modifications regardless of pressure. Think of pressure as a structural constraint (shape of the container) and phase as a material constraint (mutability of the contents).

13.5.2 Querying and Removing Pressure

Use `pressure_of()` to check the current pressure on a variable, and `depressurize()` to remove it:

Listing 13.18: Querying and removing pressure

```
flux items = [1, 2, 3]

pressurize("items", "no_grow")
print(pressure_of("items"))  // "no_grow"

depressurize("items")
print(pressure_of("items"))  // nil (no pressure)

items.push(4)  // Now allowed again
print(items)   // [1, 2, 3, 4]
```

13.5.3 Implementation Details

Pressure constraints are stored in the `LatRuntime` struct as an array of `RTPressure` entries (see `include/runtime.h`). Each entry maps a variable name to a mode string. The pressure check is performed by array mutating methods in the VM (e.g., during `push()`, `pop()`, `insert()`, and `remove_at()`), which look up the variable name in the pressure table and reject operations that violate the constraint.

The native functions `native_pressurize()`, `native_depressurize()`, and `native_pressure_of()` are defined in `src/runtime.c`. They validate the mode string, update the pressure table, and handle the swap-remove pattern for deregistration.

13.6 Seeds: seed(), unseed()

Seeds are *validation contracts* that guard the crystallization process. When you seed a variable, you attach a closure that must approve the value before it can be frozen. If the seed contract rejects the value, the freeze fails with an error.

Listing 13.19: Setting a seed contract

```
flux user_age = 25

seed(user_age, |val: any| {
    if val < 0 {
        return "age cannot be negative"
    }
    if val > 150 {
        return "age exceeds reasonable limit"
    }
    return nil  // nil means "approved"
})

// This freeze works: 25 passes the contract
fix sealed_age = freeze(user_age)
print(sealed_age)  // 25
```

Listing 13.20: Seed contract rejecting a freeze

```
flux invalid_score = -10

seed(invalid_score, |val: any| {
    if val < 0 {
        return "score must be non-negative"
    }
    return nil
})

// This freeze fails: -10 does not pass the contract
// fix bad = freeze(invalid_score)
// Error: score must be non-negative
```

Seed

A *seed* is a validation contract (a closure) attached to a variable. The seed is checked when the variable is frozen via `freeze()` or `grow()`. If the closure returns a non-nil value (typically an error message string), the freeze is rejected. If it returns `nil`, the freeze proceeds.

13.6.1 Seeds and grow()

The grow() function is a higher-level operation that validates all seeds and then freezes the variable. Unlike a bare freeze(), grow() *consumes* the seed contracts—after a successful grow(), the seeds are removed:

Listing 13.21: Using grow() with seeds

```
flux balance = 1000

seed(balance, |val: any| {
    if val < 0 { return "balance cannot be negative" }
    return nil
})

// grow() validates seeds, freezes, and removes the seed contracts
fix final_balance = grow("balance")
print(final_balance)  // 1000
// The seed contract is now consumed
```

The grow() function is implemented as native_grow() in src/runtime.c. It calls rt_validate_seeds() with the consume flag set to true, which both validates the contracts and removes them from the runtime's seed table upon success. After validation, it freezes the value, records the phase change in the tracking history, fires any cascade bonds, and triggers reactions.

13.6.2 Removing Seeds Manually

unseed() removes the seed contract for a variable without triggering any validation:

Listing 13.22: Removing a seed contract

```
flux data = "sensitive"

seed(data, |val: any| {
    if len(val) < 8 { return "too short" }
    return nil
})

// Changed our mind---remove the seed
unseed(data)

// Now freeze works without validation
fix sealed = freeze(data)
```

13.6.3 Multiple Seeds

You can attach multiple seed contracts to the same variable. All of them must approve the value for the freeze to succeed:

Listing 13.23: Multiple seed contracts

```
flux password = "hunter2"

seed(password, |val: any| {
    if len(val) < 8 { return "password must be at least 8 characters" }
    return nil
})

seed(password, |val: any| {
    if val == "password" { return "password cannot be 'password'" }
    return nil
})

// "hunter2" is only 7 characters---first seed rejects it
// fix sealed = freeze(password)
// Error: password must be at least 8 characters
```

13.7 Building Reactive Data Flows

By combining bonds, reactions, seeds, and pressure, you can build sophisticated reactive data flows where phase transitions cascade through your data model automatically.

13.7.1 Example: Configuration Validation Pipeline

Here is a complete example that uses seeds to validate configuration, bonds to cascade freezing from a primary config to dependent caches, and reactions to log all phase changes:

Listing 13.24: A reactive configuration pipeline

```
// Set up the primary configuration
flux app_config = Map::new()
app_config.set("database_url", "postgres://localhost/mydb")
app_config.set("cache_ttl", "300")
app_config.set("max_retries", "3")

// Set up a dependent cache configuration
flux cache_config = Map::new()
cache_config.set("ttl", "300")
cache_config.set("backend", "redis")

// Bond: when app_config freezes, cache_config freezes too
bond(cache_config, "app_config", "mirror")

// Seed: validate configuration before freezing
seed(app_config, |val: any| {
    if val.get("database_url") == nil {
        return "database_url is required"
    }
    return nil
})

// Reaction: log phase changes
react(app_config, |phase: any| {
    print("app_config phase -> " + phase)
})

react(cache_config, |phase: any| {
    print("cache_config phase -> " + phase)
})

// Freeze the primary config
// 1. Seed validates (database_url exists) -> passes
// 2. app_config freezes
// 3. Reaction fires: "app_config phase -> crystal"
// 4. Bond cascade: cache_config freezes
// 5. Reaction fires: "cache_config phase -> crystal"
fix sealed_config = freeze(app_config)

print(phase_of(app_config))    // "crystal"
print(phase_of(cache_config))  // "crystal"
```

13.7.2 Example: Sensor Data with Pressure Guards

Listing 13.25: Sensor data with pressure and seeds

```
flux readings = []

// Pressure: this buffer only grows (no accidental data loss)
pressurize("readings", "no_shrink")

// Seed: require at least 10 readings before allowing freeze
seed(readings, |val: any| {
    if len(val) < 10 {
        return "need at least 10 readings before sealing"
    }
    return nil
})

// Collect sensor data
for i in 0..15 {
    readings.push(20.0 + to_float(i) * 0.5)
}

// readings.pop()  // Error: pressure constraint (no_shrink)

// Now we have 15 readings, seed contract will pass
fix sealed_readings = freeze(readings)
print(sealed_readings.len())  // 15
```

13.7.3 Example: Multi-Stage Initialization

Listing 13.26: Multi-stage initialization with bonds

```
flux database = Map::new()
flux http_server = Map::new()
flux app = Map::new()

// Set up cascading bonds
bond(http_server, "database", "mirror")
bond(app, "http_server", "mirror")

// Reactions for monitoring
react(database, |phase: any| {
    print("Database: " + phase)
})
react(http_server, |phase: any| {
    print("HTTP Server: " + phase)
})
react(app, |phase: any| {
    print("Application: " + phase)
})

// Configure each component
database.set("host", "localhost")
database.set("port", "5432")

http_server.set("port", "8080")
http_server.set("workers", "4")

app.set("name", "MyApp")
app.set("version", "1.0.0")

// Freeze the database layer
// This cascades: database -> http_server -> app
freeze(database)
// Output:
//   Database: crystal
//   HTTP Server: crystal
//   Application: crystal

print(phase_of(app))  // "crystal"
```

Cascade Ordering

Bond cascades propagate synchronously during the freeze operation. If you have a chain of bonds (A → B → C), freezing A will freeze B, which will then freeze C, all within the same operation. Be careful with long chains—each step involves a full freeze (deep clone into an arena).

13.8 Exercises

1. **Alloy Design.** Design an alloy struct for a bank account with immutable fields for the account number and holder name, and mutable fields for the balance and transaction history. Write functions to deposit, withdraw, and display the account. Ensure the account number can never be changed.

2. **Reactive Counter.** Create two variables, `counter` and `display`. Set up a reaction on `counter` that prints the phase change, and a bond from `display` to `counter` using the mirror strategy. Freeze `counter` and verify that `display` freezes automatically.

3. **Seed Validation.** Write a seed contract for an email address variable that validates the value contains an `@` character and has at least 5 characters. Test the seed by trying to freeze valid and invalid email addresses.

4. **Pressure Experiment.** Create an array under `"no_grow"` pressure. Verify that `push()` fails but element assignment works. Then depressurize and confirm `push()` works again. Switch to `"no_resize"` and verify both `push()` and `pop()` fail.

5. **Complete Pipeline.** Build a complete reactive pipeline with three variables: `raw_data`, `validated_data`, and `archived_data`. Use seeds to validate `raw_data` before freezing, bonds to cascade the freeze to `validated_data` and then `archived_data`, and reactions to log every phase transition.

What's Next. We have explored the full range of Lattice's phase system: from basic `flux`/`fix` declarations through alloy structs, reactive bonds, and validation seeds. But all of this has operated in *casual* mode, where the runtime catches phase violations dynamically. In Chapter 14, we explore strict mode and the static phase checker—a compile-time analysis that catches phase errors before your code ever runs.

Chapter 14

Strict Mode and Static Phase Checking

Throughout this part of the book, we have explored the phase system as a runtime concept: values carry phase tags, `freeze()` changes them, and the VM checks them on every mutation. But runtime checks, by definition, happen *while your program is running*. A phase violation in rarely-executed code might lurk undetected for months.

Lattice offers a way to catch these errors earlier: *strict mode*. By adding `#mode strict` at the top of a file, you activate a static phase checker that analyzes your code *before* execution and rejects programs that violate phase rules. Strict mode does not eliminate runtime phase checks—it adds a layer of compile-time verification on top of them.

In this chapter, we explore what strict mode changes, how the static phase checker works, how to use phase annotations and constraints, and how to design APIs that take full advantage of phase-aware overloading.

14.1 `#mode strict` — What Changes

To enable strict mode, add a mode directive at the very beginning of your file:

Listing 14.1: Enabling strict mode

```
#mode strict

flux counter = 0
fix name = freeze("Lattice")
```

The directive is processed by the lexer (`src/lexer.c`) as a `TOK_MODE_DIRECTIVE` token and stored in the program's AST as the `AstMode` field (either `MODE_CASUAL` or `MODE_STRICT`, defined in `include/ast.h`). If no directive is present, the default is `MODE_CASUAL`.

When strict mode is active, the following rules are enforced by the static phase checker before any code executes:

1. **No `let` bindings.** Every variable must use `flux` or `fix`. The phase checker rejects `let` with:

   ```
   strict mode: use 'flux' or 'fix' instead of 'let' for binding 'x'
   ```

2. **No assigning to crystal bindings.** If a variable was declared with `fix` (or its expression was determined to be crystal), assignment to it is an error:

   ```
   strict mode: cannot assign to crystal binding 'config'
   ```

3. **No redundant freeze or thaw.** Freezing an already-crystal value or thawing an already-fluid value is flagged:

   ```
   strict mode: cannot freeze an already crystal value
   strict mode: cannot thaw an already fluid value
   ```

4. **No phase mismatch on binding.** Binding a crystal expression with `flux` is an error:

   ```
   cannot bind crystal value with flux for 'x'
   ```

5. **No fluid values in `spawn`.** Referencing a fluid variable inside a `spawn` block is rejected:

   ```
   strict mode: cannot use fluid binding 'counter' across thread boundary in
   spawn
   ```

6. **Destructuring requires explicit phase.** Destructuring bindings must specify `flux` or `fix`.

7. **Phase annotations are validated.** `@fluid` and `@crystal` annotations are checked against the initializer's actual phase.

8. **Function arguments checked against parameter phases.** If a function declares parameter phases, the checker verifies that call-site arguments have compatible phases.

Strict Mode Is Per-File

The `#mode strict` directive applies to the file where it appears. You can have strict-mode library code called by casual-mode application code, or vice versa. The phase checker runs independently on each file during compilation.

14.2 The Static Phase Checker

The static phase checker is implemented in `src/phase_check.c`. It is a single-pass abstract interpreter that walks the AST, tracking the phase of every binding in a scope stack, and emitting errors when phase rules are violated.

14.2.1 Architecture

The checker maintains a `PhaseChecker` struct with:

- `mode` — the `AstMode` (`MODE_CASUAL` or `MODE_STRICT`).
- `errors` — a vector of error message strings.
- `scope_stack` — a stack of hash maps (`LatMap`), each mapping variable names to their `AstPhase` (the compile-time phase annotation, not the runtime tag).
- `struct_defs` — a map of struct declarations for looking up field types.
- `fn_defs` — a map of function declarations for checking call-site argument phases.

14.2.2 Scope Tracking

The checker uses a scope stack to track variable phases through nested blocks. When a new scope begins (function body, loop body, if-branch, block expression), `pc_push_scope()` creates a fresh map. When the scope ends, `pc_pop_scope()` discards it. Variable lookup (`pc_lookup()`) searches from the innermost scope outward, mimicking runtime variable resolution:

Listing 14.2: Phase tracking through scopes

```
#mode strict

flux outer = 10

if outer > 5 {
    fix inner = freeze(outer)
    // Phase checker knows: outer=fluid, inner=crystal
    // inner = 20  // Error: cannot assign to crystal binding
}
// Phase checker knows: inner is out of scope now
```

14.2.3 Expression Phase Inference

The function `pc_check_expr()` recursively analyzes each expression and returns its inferred phase:

- **Literals** (integers, floats, strings, booleans, nil): return `PHASE_UNSPECIFIED`.
- **Identifiers**: return the phase registered in the scope stack via `pc_lookup()`.
- **Freeze expressions**: check the inner expression, then return `PHASE_CRYSTAL`. In strict mode, if the inner expression is already crystal, emit an error.
- **Thaw expressions**: return `PHASE_FLUID`. In strict mode, if the inner expression is already fluid, emit an error.
- **Clone expressions**: return the same phase as the inner expression.
- **Tuple expressions**: always return `PHASE_CRYSTAL` (tuples are inherently immutable).
- **Field access**: if the object is crystal, the accessed field is also crystal.
- **Forge blocks**: always return `PHASE_CRYSTAL`.
- **Anneal expressions**: always return `PHASE_CRYSTAL`.

14.2.4 Statement Checking

The function `pc_check_stmt()` handles bindings, assignments, loops, and other statements:

- **Bindings**: In strict mode, `PHASE_UNSPECIFIED` (i.e., `let`) triggers an error. The phase of the binding is recorded: `PHASE_FLUID` for `flux`, `PHASE_CRYSTAL` for `fix`. Cross-phase binding (e.g., crystal value into `flux` binding) is flagged.
- **Assignments**: In strict mode, if the target identifier is crystal, the assignment is rejected.
- **Import statements**: Imported bindings are registered as `PHASE_UNSPECIFIED`.

14.2.5 The Spawn Checker

Strict mode includes a specialized checker for `spawn` blocks: `pc_check_spawn_expr()` and `pc_check_spawn_stmt()` in `src/phase_check.c`. These functions walk the spawn body and flag any reference to a fluid binding from an outer scope:

Listing 14.3: The spawn checker in action

```
#mode strict

flux shared = [1, 2, 3]
fix safe = freeze([4, 5, 6])

scope {
    spawn {
        // print(shared)  // Error: cannot use fluid binding
        //                //        'shared' in spawn
        print(safe)        // OK: safe is crystal
    }
}
```

This check catches potential data races at compile time. In casual mode, the same code would run and might produce incorrect results if the fluid value were modified concurrently. Strict mode prevents this entire class of bugs.

14.3 Phase Annotations: `@fluid`, `@crystal`

Phase annotations are optional markers that assert the expected phase of a binding or function. They are placed before the declaration using the `@` symbol:

Listing 14.4: Phase annotations on variable bindings

```
#mode strict

@crystal
fix config = freeze(Map::new())

@fluid
flux counter = 0
```

14.3.1 Annotation Syntax

The parser recognizes two annotations: `@fluid` and `@crystal`. They can appear before:

- **Variable bindings**: `@crystal fix x = freeze(42)`

- **Function declarations**: `@fluid fn process(data: any) { ... }`

Any other annotation name (e.g., `@readonly`, `@mutable`) produces a parse error:

```
unknown annotation '@readonly' (expected @fluid or @crystal)
```

If an annotation appears before a construct that is not a function or binding, the parser also rejects it:

```
@fluid/@crystal annotation must precede fn or variable binding
```

14.3.2 What Annotations Do

Annotations are *assertions*, not directives. They do not change the phase of the value—they verify that the value's phase matches the annotation. The phase checker validates annotations against the initializer's inferred phase:

Listing 14.5: Annotation violation

```
#mode strict

@crystal
flux counter = 0
// Error: @crystal annotation violated: initializer for 'counter' is fluid
```

The reverse also triggers an error:

Listing 14.6: Reverse annotation violation

```
#mode strict

@fluid
fix data = freeze([1, 2, 3])
// Error: @fluid annotation violated: initializer for 'data' is crystal
```

14.3.3 Annotations on Functions

When applied to a function, a phase annotation indicates the expected phase behavior of the function's return value or its overall role in the phase system:

Listing 14.7: Annotated functions

```
#mode strict

@crystal
fn default_config() {
    return freeze(Map::new())
}

@fluid
fn create_buffer(size: any) {
    flux buf = []
    for i in 0..size {
        buf.push(0)
    }
    return buf
}
```

These annotations serve as documentation and contract: they signal to readers (and to the phase checker) the function's intended phase behavior.

Annotations as Documentation

Even in casual mode, phase annotations are valuable as documentation. They make the author's intent explicit without requiring strict mode's full enforcement. Consider adding them to public API functions in library code.

14.4 Phase Constraints: `(~| *)`, `(flux| fix)`

While annotations assert a single phase, *phase constraints* on function parameters specify which phases are acceptable for an argument. Constraints use a compact syntax inside parentheses, with the pipe symbol | separating alternatives:

Listing 14.8: Phase constraints on parameters

```
#mode strict

fn display_value(data: (~|*) any) {
    // data can be either fluid (~) or crystal (*)
    print(data)
}

fn read_only_process(data: (*) any) {
    // data must be crystal
    print("Processing: " + to_string(data))
}
```

14.4.1 Constraint Syntax

Phase constraints are parsed by the type expression parser in `src/parser.c`. Inside parentheses, the following symbols are recognized:

Symbol	Meaning
`~` or `flux`	Accepts fluid values (`PCON_FLUID`)
`*` or `fix`	Accepts crystal values (`PCON_CRYSTAL`)
`sublimated`	Accepts sublimated values (`PCON_SUBLIMATED`)

Table 14.1: Phase constraint symbols.

Constraints are represented internally as a bitmask (`PhaseConstraint` type, defined in `include/ast.h`):

- `PCON_FLUID` = `0x01`
- `PCON_CRYSTAL` = `0x02`
- `PCON_SUBLIMATED` = `0x04`
- `PCON_ANY` = `0x07` (all bits set)

Multiple symbols are combined with the pipe (`|`) separator:

Listing 14.9: Composite phase constraints

```
#mode strict

// Accept fluid or crystal, but not sublimated
fn flexible_handler(val: (~|*) any) {
    print(to_string(val))
}

// Accept only crystal or sublimated (immutable variants)
fn immutable_handler(val: (*|sublimated) any) {
    print(to_string(val))
}
```

14.4.2 Constraint Checking

The function `constraint_accepts()` in `src/phase_check.c` checks whether a given argument phase satisfies a constraint bitmask:

- If the constraint is 0 (no constraint specified), any phase is accepted.
- Otherwise, the argument's phase must have its corresponding bit set in the bitmask.
- `PHASE_UNSPECIFIED` arguments are always accepted (they are compatible with any constraint).

In strict mode, the phase checker verifies constraints at every call site:

Listing 14.10: Constraint violation in strict mode

```
#mode strict

fn process_frozen(data: (*) any) {
    print(data)
}

flux mutable_data = [1, 2, 3]
// process_frozen(mutable_data)
// Error: strict mode: no matching overload for 'process_frozen'
//        with given argument phases
```

Phase Constraint

A *phase constraint* is a bitmask attached to a function parameter's type that specifies which value phases are acceptable for that argument. Constraints are checked at compile time in strict mode and during overload resolution at runtime.

14.5 Phase-Dependent Function Overloading

One of Lattice's most distinctive features is *phase-dependent function overloading*: you can define multiple versions of a function with the same name but different parameter phase constraints, and the runtime (or the strict-mode checker) selects the appropriate one based on the phases of the actual arguments.

Listing 14.11: Phase-dependent overloading

```
fn process(data: flux any) {
    print("Processing mutable data in-place")
    data.push(0)
}

fn process(data: fix any) {
    print("Processing frozen data (creating a copy)")
    flux copy = thaw(data)
    copy.push(0)
    return freeze(copy)
}

flux live_data = [1, 2, 3]
fix frozen_data = freeze([4, 5, 6])

process(live_data)    // Calls the flux overload
process(frozen_data)  // Calls the fix overload
```

14.5.1 How Overload Resolution Works

Functions with the same name are chained together in a linked list via the `next_overload` pointer in the `FnDecl` struct (`include/ast.h`). When a function is called, the runtime walks this chain to find the best matching overload.

The `resolve_overload()` function in `src/eval.c` implements the resolution algorithm:

1. For each overload candidate, check **arity compatibility**: does the argument count match the parameter count (accounting for default values and variadic parameters)?
2. For each argument, check **phase compatibility**: is the argument's runtime phase compatible with the parameter's declared phase?
3. Compute a **score** for the match. Exact phase matches score highest (3 points each), compatible-but-unspecified matches score lower (1–2 points).
4. Select the candidate with the highest total score. If no candidate is compatible, emit an error:

```
no matching overload for 'process' with given argument phases
```

The scoring system ensures that more specific overloads are preferred. An overload declaring `flux` for a parameter will beat one declaring no phase (unspecified) when passed a fluid argument.

14.5.2 Overloading in Strict Mode

In strict mode, the phase checker performs a compile-time version of overload resolution. When a call expression targets a function name that has multiple overloads, the checker:

1. Infers the phase of each argument expression.
2. Tests each overload's parameter constraints against the inferred argument phases.
3. If no overload matches, emits:

```
strict mode: no matching overload for 'process' with given argument phases
```

This means phase-mismatch errors are caught before the program runs:

Listing 14.12: Compile-time overload checking

```
#mode strict

fn handle(data: flux any) {
    data.push(0)
}

fix frozen = freeze([1, 2, 3])
// handle(frozen)
// Error: strict mode: no matching overload for 'handle'
//        with given argument phases
```

14.5.3 Registration and Chaining

When the evaluator or compiler encounters multiple function declarations with the same name, it registers them using `register_fn_overload()` (in `src/eval.c` for the tree-walker, and `pc_register_fn()` in `src/phase_check.c` for the static checker).

Two functions with the same name are considered different overloads if their *phase signatures* differ. The function `phase_signatures_match()` compares parameter counts and per-parameter phases:

Listing 14.13: Different phase signatures create overloads

```
// These are two different overloads:
fn serialize(data: flux any) { /* ... */ }
fn serialize(data: fix any) { /* ... */ }

// These have the SAME phase signature (both unspecified):
fn parse(input: any) { /* version 1 */ }
fn parse(input: any) { /* version 2: replaces version 1 */ }
```

If two functions have identical phase signatures, the second definition *replaces* the first (it is not treated as a new overload). Only functions with *different* phase signatures coexist in the overload chain.

Overload Ambiguity

If an argument is `VTAG_UNPHASED` (e.g., from a `let` binding in casual mode), it is compatible with both fluid and crystal overloads. The resolver picks the one with the highest score, but if scores are tied, the last-registered overload wins. In strict mode, `let` is forbidden, so unphased ambiguity does not arise.

14.6 Designing APIs That Are Phase-Aware

The phase system is most powerful when your API boundaries are designed with phases in mind. Here are patterns and guidelines for building phase-aware libraries and modules.

14.6.1 Produce Crystal, Accept Any

A common pattern: functions that return data should return it frozen (crystal), while functions that accept data should be flexible about the input phase:

Listing 14.14: Produce crystal, accept any

```
fn get_defaults() {
    // Always return frozen defaults
    return forge {
        flux temp = Map::new()
        temp.set("timeout", "30")
        temp.set("retries", "3")
        freeze(temp)
    }
}

fn apply_config(base: any, overrides: any) {
    // Accept any phase, thaw if needed
    flux working = thaw(base)
    // Apply overrides...
    return freeze(working)
}
```

14.6.2 Phase-Specific Behavior via Overloading

Use overloading to provide optimized paths for different phases:

Listing 14.15: Optimized paths via phase overloading

```
fn update_score(scores: flux Array, index: any, delta: any) {
    // Fluid path: modify in place (efficient)
    scores[index] = scores[index] + delta
    return scores
}

fn update_score(scores: fix Array, index: any, delta: any) {
    // Crystal path: create modified copy
    flux copy = thaw(scores)
    copy[index] = copy[index] + delta
    return freeze(copy)
}
```

Callers do not need to know which overload is selected—the function handles both cases correctly.

14.6.3 Guard Clauses with `phase_of()`

For functions that need phase-dependent behavior in casual mode (where overloading may not be strictly enforced), use runtime phase checks as guard clauses:

Listing 14.16: Runtime phase guards

```
fn safe_update(collection: any, key: any, value: any) {
    if phase_of(collection) == "crystal" {
        print("Warning: cannot update crystal collection")
        return collection
    }
    if phase_of(collection) == "sublimated" {
        print("Error: collection is permanently sealed")
        return collection
    }
    collection.set(key, value)
    return collection
}
```

14.6.4 Strict Mode in Library Code

Consider enabling strict mode in library code even if your application code uses casual mode. Libraries are consumed by many callers, and phase errors in library code can be particularly hard to debug:

Listing 14.17: Strict mode for library safety

```
// lib/cache.lat
#mode strict

@crystal
fn create_cache(max_size: any) {
    return forge {
        flux temp = Map::new()
        temp.set("__max_size", to_string(max_size))
        freeze(temp)
    }
}

fn cache_get(cache: fix any, key: any) {
    return cache.get(key)
}
```

14.6.5 Documenting Phase Expectations

Even without strict mode, use comments and naming conventions to signal phase expectations:

Listing 14.18: Phase documentation conventions

```
// Convention: functions returning frozen data end with _frozen
fn load_config_frozen(path: any) {
    // ... load and return frozen config ...
    return freeze(config)
}

// Convention: functions taking mutable data start with mutate_
fn mutate_append(list: any, item: any) {
    list.push(item)
}
```

Phase-Aware API Checklist

When designing a public API, ask these questions:

1. Should returned values be frozen? (Usually yes, for safety.)
2. Do different parameter phases warrant different behavior? (Consider overloading.)
3. Should the function reject certain phases? (Use constraints or runtime guards.)
4. Is this code safety-critical enough for strict mode? (Libraries: probably yes.)

14.7 Putting It All Together: A Strict-Mode Example

Let us write a complete strict-mode program that demonstrates phase annotations, constraints, overloading, and the spawn checker:

Listing 14.19: A complete strict-mode program

```
#mode strict

// Phase-annotated configuration builder
@crystal
fn build_config(host: any, port: any) {
    return forge {
        flux temp = Map::new()
        temp.set("host", host)
        temp.set("port", to_string(port))
        temp.set("started_at", "2024-01-15T10:00:00Z")
        freeze(temp)
    }
}

// Phase-dependent overloads for processing
fn process_request(config: fix any, path: any) {
    fix response = freeze("200 OK: " + path)
    return response
}

// Entry point
fn main() {
    fix config = build_config("0.0.0.0", 8080)

    // config is crystal---safe to share across spawn boundaries
    fix paths = freeze(["/api/health", "/api/users", "/api/data"])

    scope {
        for path in paths {
            spawn {
                // All bindings here are crystal---strict mode is happy
                fix result = process_request(config, path)
                print(result)
            }
        }
    }
}
```

This program compiles without any strict-mode errors because:

- Every binding uses `flux` or `fix` (no `let`).

- Only crystal values (`config`, `paths`) cross the spawn boundary.
- The `process_request` function's parameter constraint matches the crystal argument.
- Phase annotations on `build_config` match its forge-block return (crystal).

14.8 Exercises

1. **Strict Mode Conversion.** Take any program you have written while following this book and add `#mode strict` at the top. Fix all the phase errors the checker reports. How many `let` bindings did you have to change to `flux` or `fix`?
2. **Phase-Overloaded API.** Write a function `sort_data` with two overloads: one for fluid arrays (sorts in place) and one for crystal arrays (returns a new sorted frozen array). Test both overloads and verify the correct one is called.
3. **Constraint Design.** Write a function that accepts only crystal or sublimated values using the (*sublimated)| constraint syntax. Test it with fluid, crystal, and sublimated arguments. Verify that the fluid argument is rejected in strict mode.
4. **Spawn Safety.** Write a strict-mode program with a scope containing multiple spawn blocks. Try to reference a fluid variable from within a spawn and observe the compile-time error. Then fix the program by freezing the shared data before the scope.

What's Next. With the phase system fully explored—from basic `flux`/`fix` declarations through memory arenas, alloy structs, reactive bonds, and strict-mode static checking—we are ready to move beyond values and into the world of composite types. In Part IV, we explore how Lattice's struct, enum, trait, and generic systems interact with the phase system you now understand, giving you the tools to build sophisticated, type-safe abstractions.

Part IV

Structures and Abstractions

Chapter 15

Structs

Every interesting program eventually needs to group related data together. An array of coordinates, a pair of first-and-last names, a record that bundles a user's email with their role—the moment your data grows beyond a single value, you reach for a *struct*.

Lattice structs are lightweight, named containers for fields. They are pass-by-value, they support methods through trait implementations and callable fields, and they integrate naturally with the phase system you met in Chapter 11. In this chapter we will define structs, read and write their fields, attach behavior, customise equality, and peer under the hood with the reflection API.

15.1 Defining and Instantiating Structs

A struct definition gives a name and a list of typed fields:

Listing 15.1: A minimal struct definition

```
struct Point { x: any, y: any }
```

The keyword `struct` is followed by a name that must begin with an uppercase letter, then a pair of braces enclosing comma-separated fields. Each field has a name and a type annotation. Right now the most common annotation is `any`, which accepts every type; we will explore richer type annotations when we reach generics in Chapter 18.

Struct

A *struct* is a named, composite value type whose data is stored in a fixed set of named fields. Structs are **pass-by-value**: assigning a struct to a new variable produces an independent copy.

Once defined, you create an instance with a *struct literal*—the struct name followed by braces and field initialisers:

Listing 15.2: Creating struct instances

```
struct Point { x: any, y: any }

let origin = Point { x: 0, y: 0 }
let target = Point { x: 42, y: 17 }

print(origin)  // Point { x: 0, y: 0 }
print(target)  // Point { x: 42, y: 17 }
```

Every field must be supplied when constructing an instance—there are no default values at the language level. If you want defaults, write a factory function:

Listing 15.3: A factory function with defaults

```
struct Config {
    host: String,
    port: Int,
    verbose: Bool
}

fn default_config() -> Config {
    return Config {
        host: "localhost",
        port: 8080,
        verbose: false
    }
}

let cfg = default_config()
print(cfg.port)  // 8080
```

15.1.1 Nested Structs

Fields can hold any value, including other structs:

Listing 15.4: Nested structs

```
struct Point { x: any, y: any }
struct Rect { origin: any, width: any, height: any }

let r = Rect {
    origin: Point { x: 5, y: 10 },
    width: 100,
    height: 50
}

print(r.origin.x)  // 5
print(r.width)     // 100
```

Nesting is a natural way to compose data. Because structs are pass-by-value, the `Point` inside the `Rect` is a full copy, not a reference. Modifying the original `Point` after constructing the `Rect` has no effect on `r.origin`.

15.1.2 Structs with Typed Fields

Although `any` works everywhere, you can annotate fields with concrete types for documentation and, in strict mode, static checking:

Listing 15.5: Typed struct fields

```
struct Entity {
    id: Int,
    name: String,
    x: Int,
    y: Int,
    hp: Int,
    max_hp: Int,
    tag: String
}
```

This reads like a blueprint: an `Entity` carries an integer ID, a name, a position, health, and a tag string. Readers of your code—and the compiler in strict mode—will thank you for the specificity.

15.2 Field Access and Mutation

Reading a field uses dot notation:

Listing 15.6: Field access

```
struct Point { x: any, y: any }

let p = Point { x: 3, y: 4 }
print(p.x)      // 3
print(p.y)      // 4
print(p.x + p.y) // 7
```

15.2.1 Mutation Requires `flux`

Because structs follow the phase system, a struct bound with `let` (or `fix`) is *crystallised*—its fields cannot be changed. To mutate fields, bind the struct with `flux`:

Listing 15.7: Mutating struct fields

```
struct Point { x: any, y: any }

flux p = Point { x: 1, y: 2 }
print(p.x)  // 1

p.x = 99
print(p.x)  // 99

p.y = 88
print(p.y)  // 88
```

Compound assignment operators work on fields too:

Listing 15.8: Compound field assignment

```
struct Point { x: any, y: any }

flux p = Point { x: 10, y: 20 }
p.x += 5
print(p.x)  // 15
p.y -= 10
print(p.y)  // 10
p.x *= 2
print(p.x)  // 30
```

Mutation is Shallow

When you mutate a field on a `flux` struct, you are updating *that copy's* field. Because structs are pass-by-value, any other variable that received a copy of the struct is unaffected. This is one of the most important mental-model points in Lattice. We will return to it in Section 15.8.

15.2.2 Optional Field Access

If you have a value that might be `nil`, the optional-chaining operator `?.` lets you access fields safely:

Listing 15.9: Optional field access

```
struct User { name: any, email: any }

fn find_user(id: any) -> any {
    if id == 1 {
        return User { name: "Alice", email: "alice@example.com" }
    }
    return nil
}

let user = find_user(42)
print(user?.name)   // nil (no crash)
print(user?.email)  // nil
```

If the receiver is `nil`, the entire expression evaluates to `nil` instead of producing a runtime error. This pairs well with the `??` nil-coalescing operator covered in Chapter 5.

15.2.3 Passing Structs to Functions

Structs can be passed as function arguments and returned from functions. Remember: each pass produces an independent copy.

Listing 15.10: Structs as arguments and return values

```
struct Point { x: any, y: any }

fn sum_point(pt: any) -> any {
    return pt.x + pt.y
}

fn make_point(a: any, b: any) -> Point {
    return Point { x: a, y: b }
}

let p = Point { x: 3, y: 4 }
print(sum_point(p))  // 7

let q = make_point(100, 200)
print(q.x)  // 100
print(q.y)  // 200
```

15.2.4 Structs in Collections

Arrays, maps, and sets can all hold structs:

Listing 15.11: An array of structs

```
struct Point { x: any, y: any }

let points = [
    Point { x: 1, y: 2 },
    Point { x: 3, y: 4 },
    Point { x: 5, y: 6 }
]

for pt in points {
    print(pt.x + pt.y)
}
// Output: 3, 7, 11
print(len(points))  // 3
```

15.3 Methods via Traits: `impl` Blocks

Lattice attaches behaviour to structs through *trait implementations*. A trait declares a set of method signatures; an `impl` block provides the bodies for a specific struct. We will explore traits in depth in Chapter 17, but here is enough to get you started.

Listing 15.12: A trait and its implementation

```
struct Point { x: any, y: any }

trait Describable {
    fn describe(self: any) -> any
}

impl Describable for Point {
    fn describe(self: any) -> any {
        return "Point(" + to_string(self.x) + ", " + to_string(self.y) + ")"
    }
}

let p = Point { x: 7, y: 8 }
print(p.describe())  // Point(7, 8)
```

The first parameter of every method is `self`, which receives a copy of the struct instance the method was called on. Inside the body you read fields with `self.x`, `self.y`, and so on.

Methods Return New Values

Because `self` is a copy, methods that "modify" the struct must return a new instance. The original is untouched:

```
trait Scalable {
    fn scale(self: any, factor: any) -> any
}

impl Scalable for Point {
    fn scale(self: any, factor: any) -> any {
        return Point { x: self.x * factor, y: self.y * factor }
    }
}

let p = Point { x: 5, y: 10 }
let q = p.scale(3)
print(q.x)  // 15
print(q.y)  // 30
print(p.x)  // 5  (unchanged)
```

15.3.1 Multiple Traits per Struct

A single struct can implement as many traits as you like:

Listing 15.13: A struct implementing multiple traits

```
struct Point { x: any, y: any }

trait HasArea {
    fn area(self: any) -> any
}

trait Describable {
    fn describe(self: any) -> any
}

impl HasArea for Point {
    fn area(self: any) -> any {
        return self.x * self.y
    }
}

impl Describable for Point {
    fn describe(self: any) -> any {
        return "Point(" + to_string(self.x) + ", " + to_string(self.y) + ")"
    }
}

let p = Point { x: 3, y: 4 }
print(p.area())      // 12
print(p.describe())  // Point(3, 4)
```

Under the hood, the compiler registers each method with a key of the form `TypeName::method_name` in the global scope. When you call `p.describe()`, the VM looks up `Point::describe` and invokes it with `self` bound to a clone of `p`. You can see how method dispatch is compiled in `src/stackcompiler.c`—look for the `OP_INVOKE` family of opcodes.

15.3.2 Methods in Loops

Methods compose naturally with iteration:

Listing 15.14: Calling methods in a loop

```
struct Point { x: any, y: any }

trait HasArea {
    fn area(self: any) -> any
}

impl HasArea for Point {
    fn area(self: any) -> any {
        return self.x * self.y
    }
}

let points = [
    Point { x: 1, y: 1 },
    Point { x: 2, y: 3 },
    Point { x: 4, y: 5 }
]

flux total = 0
for p in points {
    total = total + p.area()
}
print(total)  // 1 + 6 + 20 = 27
```

15.4 Callable Struct Fields

Besides attaching methods through traits, you can embed closures directly as struct fields. Any field whose value is a closure becomes *callable*: when you invoke it with dot syntax, Lattice finds the closure field and calls it.

Listing 15.15: A struct with a callable closure field

```
struct Greeter {
    greeting: String,
    greet: Fn
}

let hello = Greeter {
    greeting: "Hello",
    greet: |self| {
        self.greeting + ", world!"
    }
}

print(hello.greet())  // Hello, world!
```

The closure receives `self` as its first argument when called through dot notation, just like a trait method. The difference is that the closure is stored *inside* the struct value itself—it travels with the data.

15.4.1 Stateful Callable Fields

Callable fields shine when you want per-instance behaviour. Consider a vending machine where each operation is a closure:

Listing 15.16: Vending machine with callable fields

```
struct VendingMachine {
    balance: Int,
    inventory: Map,
    state: String,
    insert_coin: Fn,
    select_item: Fn,
    dispense: Fn,
    display: Fn
}

fn make_vending_machine() -> VendingMachine {
    flux inv = Map::new()
    inv.set("cola", 150)
    inv.set("chips", 100)
    inv.set("candy", 75)

    return VendingMachine {
        balance: 0,
        inventory: inv,
        state: "idle",
        insert_coin: |self, amount| {
            self.balance + amount
        },
        select_item: |self, item| {
            if !self.inventory.has(item) {
                "error: unknown item"
            } else {
                let price = self.inventory.get(item)
                if self.balance < price {
                    "error: insufficient funds"
                } else {
                    "ok:" + item
                }
            }
        },
        dispense: |self, item| {
            let price = self.inventory.get(item)
            let change = self.balance - price
            "Dispensing " + item + "! Change: " + to_string(change) + " cents"
        },
        display: |self| {
            "Balance: " + to_string(self.balance) + " cents"
        }
    }
}
```

Each closure can read `self.balance` and `self.inventory`. The caller updates the struct's mutable state after each operation:

Listing 15.17: Driving the vending machine

```
flux vm = make_vending_machine()
print(vm.display())  // Balance: 0 cents

vm.balance = vm.insert_coin(25)
vm.balance = vm.insert_coin(25)
vm.balance = vm.insert_coin(25)
print(vm.display())  // Balance: 75 cents

let result = vm.select_item("candy")
print(result)  // ok:candy
print(vm.dispense("candy"))  // Dispensing candy! Change: 0 cents
```

Trait Methods vs. Callable Fields

Use **trait methods** when the behaviour is shared across multiple structs that implement the same interface. Use **callable fields** when the behaviour is unique to each *instance*, or when you want to swap behaviour at runtime by assigning a different closure to the field.

15.4.2 How the Runtime Resolves Calls

When you write `vm.display()`, the runtime checks two places:

1. **Callable field**: Does the struct have a field named `display` whose value is a closure? If so, call it, passing the struct as `self`.
2. **Trait method**: Is there an impl method registered as `VendingMachine::display`? If so, call that with a clone of the struct as `self`.

Callable fields take priority. If a struct has both a closure field named `area` and a trait method `area`, the closure field wins. You can inspect how this dispatch works in `src/eval.c`—the method-call handler first scans `field_names` for a matching closure before falling through to the impl registry.

15.5 Custom Equality with eq

By default, two structs are equal if they have the same name and all their fields are recursively equal:

Listing 15.18: Default struct equality

```
struct Point { x: any, y: any }

let a = Point { x: 1, y: 2 }
let b = Point { x: 1, y: 2 }
let c = Point { x: 3, y: 4 }

print(a == b)  // true
print(a == c)  // false
print(a != c)  // true
```

This is structural equality—it compares data, not identity. Two distinct instances with the same field values are considered equal.

15.5.1 Overriding Equality with an `eq` Field

Sometimes you want a different notion of equality. Perhaps two users should be equal if they share the same ID, regardless of other fields. Lattice lets you override equality by adding a closure field named `eq`:

Listing 15.19: Custom equality via the eq field

```
struct User {
    id: Int,
    name: String,
    email: String,
    eq: Fn
}

fn make_user(id: Int, name: String, email: String) -> User {
    return User {
        id: id,
        name: name,
        email: email,
        eq: |self, other| {
            self.id == other.id
        }
    }
}

let alice1 = make_user(1, "Alice", "alice@example.com")
let alice2 = make_user(1, "Alice A.", "alice.a@example.com")
let bob = make_user(2, "Bob", "bob@example.com")

print(alice1 == alice2)  // true  (same ID)
print(alice1 == bob)     // false (different ID)
```

The `eq` closure receives two arguments: `self` (the left-hand operand) and `other` (the right-hand operand). Its return value is converted to a boolean. The `!=` operator automatically negates the result.

The eq Field Must Be a Closure

The custom equality mechanism specifically checks for a field named `eq` whose value is a `VAL_CLOSURE`. If you name a non-closure field `eq`, it will not be used for equality comparisons—the default structural equality applies instead.

15.5.2 When Custom Equality Matters

Custom equality is valuable for:

- **Identity-based comparison**: Compare by ID or key rather than every field.
- **Ignoring computed fields**: Skip over cached or derived data that should not affect equality.

- **Tolerance-based comparison**: For structs containing floating-point values, you might compare with a tolerance:

Listing 15.20: Approximate equality for geometry

```
struct Coordinate {
    lat: Float,
    lng: Float,
    eq: Fn
}

fn make_coord(lat: Float, lng: Float) -> Coordinate {
    return Coordinate {
        lat: lat,
        lng: lng,
        eq: |self, other| {
            let dlat = self.lat - other.lat
            let dlng = self.lng - other.lng
            if dlat < 0 { dlat = 0.0 - dlat }
            if dlng < 0 { dlng = 0.0 - dlng }
            dlat < 0.0001 && dlng < 0.0001
        }
    }
}

let a = make_coord(51.5074, -0.1278)
let b = make_coord(51.5074, -0.1278)
print(a == b)  // true
```

15.6 Struct Reflection

Lattice provides four built-in functions for inspecting and transforming structs at runtime. These live in the category of *reflection*—code that examines the shape of data programmatically.

15.6.1 `struct_name()`

Returns the type name of a struct as a string:

Listing 15.21: Getting a struct's type name

```
struct Point { x: any, y: any }
struct Rect { width: any, height: any }

let p = Point { x: 10, y: 20 }
let r = Rect { width: 5, height: 3 }

print(struct_name(p))  // "Point"
print(struct_name(r))  // "Rect"
```

This is useful for logging, serialisation, and dynamic dispatch.

15.6.2 `struct_fields()`

Returns an array of field-name strings:

Listing 15.22: Listing a struct's fields

```
struct Config {
    host: String,
    port: Int,
    verbose: Bool
}

let cfg = Config { host: "localhost", port: 8080, verbose: false }
let fields = struct_fields(cfg)
print(fields)  // ["host", "port", "verbose"]
```

The order of names matches the order in the struct definition.

15.6.3 `struct_to_map()`

Converts a struct into a `Map` where keys are field names and values are field values:

Listing 15.23: Converting a struct to a map

```
struct Point { x: any, y: any }

let p = Point { x: 42, y: 17 }
let m = struct_to_map(p)

print(m)            // {x: 42, y: 17}
print(m.get("x"))   // 42
print(m.get("y"))   // 17
```

This is the fastest way to bridge between structs and the map-based world of JSON serialisation (see **??**).

15.6.4 `struct_from_map()`

Goes the other direction—creates a struct from a type name and a map:

Listing 15.24: Creating a struct from a map

```
struct Point { x: any, y: any }

flux m = Map::new()
m.set("x", 100)
m.set("y", 200)

let p = struct_from_map("Point", m)
print(p.x)  // 100
print(p.y)  // 200
```

If a field is missing from the map, it defaults to `nil`. The struct definition must already exist in scope—`struct_from_map` looks up the registered struct metadata at runtime.

15.6.5 A Practical Reflection Example

Combining these functions, you can write a generic "debug print" utility:

Listing 15.25: A generic struct inspector

```
struct Point { x: any, y: any }
struct User { name: any, age: any }

fn inspect(val: any) -> String {
    let name = struct_name(val)
    let fields = struct_fields(val)
    let m = struct_to_map(val)
    flux parts = []
    for f in fields {
        parts.push(f + "=" + to_string(m.get(f)))
    }
    return name + "(" + parts.join(", ") + ")"
}

let p = Point { x: 3, y: 4 }
let u = User { name: "Alice", age: 30 }

print(inspect(p))  // Point(x=3, y=4)
print(inspect(u))  // User(name=Alice, age=30)
```

Reflection at Runtime

Struct reflection functions operate at *runtime*. The compiler stores field metadata as a hidden global (keyed as `__struct_StructName` in the constant pool). This means reflection incurs a small cost, but it works for any struct, even those defined in imported modules. See `src/stackcompiler.c` for how metadata is emitted during compilation.

15.7 Struct Destructuring

When you want to extract multiple fields at once, destructuring is more concise than repeated dot access:

Listing 15.26: Struct destructuring

```
struct Point { x: any, y: any }

let p = Point { x: 55, y: 66 }
let { x, y } = p
print(x)  // 55
print(y)  // 66
```

The `let { x, y } = p` syntax binds local variables `x` and `y` to the values of `p.x` and `p.y` respectively. The variable names must match the field names.

Destructuring works in any binding context where patterns are accepted, including `match` arms (see Chapter 9).

Selective Destructuring

You do not have to destructure every field. Extracting only the fields you need is perfectly valid:

```
struct Rect { origin: any, width: any, height: any }

let r = Rect { origin: Point { x: 0, y: 0 }, width: 100, height: 50 }
let { width, height } = r
print(width * height)  // 5000
```

15.8 Pass-by-Value Semantics

This is the section that will save you from the most common struct-related confusion. Lattice structs are **pass-by-value**. Every assignment, every function argument, every return value produces an *independent copy* of the struct. There are no hidden references, no shared state, no aliasing surprises.

Listing 15.27: Pass-by-value in action

```
struct Point { x: any, y: any }

flux a = Point { x: 1, y: 2 }
flux b = a          // b is a copy of a
b.x = 999

print(a.x)  // 1   (unchanged)
print(b.x)  // 999
```

After `flux b = a`, the variables `a` and `b` hold entirely separate structs. Mutating one has zero effect on the other.

15.8.1 Implications for Function Calls

When you pass a struct to a function, the function receives its own copy:

Listing 15.28: Functions receive copies

```
struct Counter { value: any }

fn try_to_increment(c: any) {
    flux local = c
    local.value = local.value + 1
    print(local.value)  // 11
}

let counter = Counter { value: 10 }
try_to_increment(counter)
print(counter.value)  // 10 (unchanged)
```

The function's local copy is incremented, but the caller's `counter` retains its original value. If you want the change to be visible to the caller, the function must *return* the new struct:

Listing 15.29: Returning the modified struct

```
struct Counter { value: any }

fn increment(c: any) -> Counter {
    return Counter { value: c.value + 1 }
}

flux counter = Counter { value: 10 }
counter = increment(counter)
print(counter.value)  // 11
```

15.8.2 Why Pass-by-Value?

You might wonder why Lattice chose value semantics when many popular languages use references. The answer ties into the phase system:

- **No aliasing bugs**: Two variables never point at the same struct, so freezing one cannot affect another.
- **Thread safety**: In concurrent code (**??**), spawned tasks receive their own copy of any struct, eliminating data races by construction.
- **Predictability**: When you see `let x = y`, you know that `x` is independent. No need to check whether the type is a "value type" or a "reference type."

Value Semantics

In Lattice, structs (and buffers) are **value types**. Assignment copies the entire struct, including all nested data. The runtime uses `value_deep_clone()` in `include/value.h` to produce the copy. For large structs this means more memory, but it guarantees isolation.

15.8.3 The `clone` Keyword

If you want to be explicit about copying—perhaps for readability in performance-sensitive code—you can use the `clone` keyword:

Listing 15.30: Explicit cloning

```
struct Point { x: any, y: any }

let original = Point { x: 1, y: 2 }
flux copy = clone original
copy.x = 42

print(original.x)  // 1
print(copy.x)      // 42
```

In practice, assignment already clones, so `clone` is primarily a documentation signal: "I know this is making a copy, and I intend that."

15.8.4 When Copies Get Expensive

For small structs like `Point`, copying is negligible. For structs with many fields or large nested arrays, the cost can add up. Strategies for managing this include:

- **Keep structs lean**: Store IDs or indices into a central collection rather than embedding large data directly.
- **Use `Ref` for shared state**: When you genuinely need shared mutable access, wrap a value in a `Ref` (covered in Chapter 11).
- **Prefer functional updates**: Return new structs from methods rather than mutating fields—the compiler can often optimise this path.

15.9 A Larger Example: Entity Component System

Let's bring everything together with a complete example. An Entity Component System (ECS) is a game architecture pattern where entities are bags of data and systems are functions that transform arrays of entities. It is a perfect showcase for struct-centric design.

Listing 15.31: An ECS in Lattice

```
struct Entity {
    id: Int,
    name: String,
    x: Int,
    y: Int,
    dx: Int,
    dy: Int,
    hp: Int,
    max_hp: Int,
    tag: String
}

fn make_entity(id: Int, name: String, tag: String) -> Entity {
    return Entity {
        id: id, name: name,
        x: 0, y: 0, dx: 0, dy: 0,
        hp: 0, max_hp: 0, tag: tag
    }
}

fn render_entity(e: Entity) {
    flux line = "  [" + to_string(e.id) + "] " + e.name
    line += " at (" + to_string(e.x) + "," + to_string(e.y) + ")"
    if e.max_hp > 0 {
        line += " HP:" + to_string(e.hp) + "/" + to_string(e.max_hp)
    }
    print(line)
}

// System: apply velocity to position
fn movement_system(entities: [Entity]) -> [Entity] {
    flux result = []
    for e in entities {
        flux moved = e
        moved.x = e.x + e.dx
        moved.y = e.y + e.dy
        result.push(moved)
    }
    return result
}
```

The movement system iterates over entities, creates updated copies with new positions, and returns the new array. Because structs are pass-by-value, each `flux moved = e` produces an independent copy that we can mutate without affecting the original.

Listing 15.32: Running the ECS simulation

```
fn main() {
    flux player = make_entity(0, "Hero", "player")
    player.x = 0
    player.y = 0
    player.dx = 1
    player.dy = 1
    player.hp = 100
    player.max_hp = 100

    flux goblin = make_entity(1, "Goblin", "enemy")
    goblin.x = 3
    goblin.y = 3
    goblin.dx = -1
    goblin.dy = -1

    flux entities = [player, goblin]

    flux tick = 0
    while tick < 3 {
        print("--- Tick " + to_string(tick) + " ---")
        for e in entities {
            render_entity(e)
        }
        entities = movement_system(entities)
        tick += 1
    }
}
```

This pattern—immutable-by-default data flowing through pure transformation functions—is idiomatic Lattice. The phase system and value semantics make it safe; the struct system makes it readable.

15.10 Exercises

1. **RGB colour struct.** Define a struct `Color` with fields `r`, `g`, and `b` (all integers 0–255). Write a function `mix(a: Color, b: Color) -> Color` that returns the average of each channel. Verify

that mixing `Color { r: 255, g: 0, b: 0 }` with `Color { r: 0, g: 0, b: 255 }` gives `Color { r: 127, g: 0, b: 127 }`.

2. **Bank account.** Create a `BankAccount` struct with fields `owner`, `balance`, and callable fields `deposit` and `withdraw`. The `withdraw` field should return an error string if the balance is insufficient. Drive a sequence of transactions and print the final balance.

3. **Custom equality.** Define a `Student` struct with fields `id`, `name`, and `gpa`. Add an `eq` closure that compares students by `id` only. Demonstrate that two `Student` instances with the same ID but different GPAs are considered equal.

4. **Reflection round-trip.** Write a function that takes any struct, converts it to a map with `struct_to_map`, doubles every numeric value in the map, and converts it back with `struct_from_map`. Test it with `Point { x: 5, y: 10 }` and verify the result is `Point { x: 10, y: 20 }`.

5. **Value semantics drill.** Without running the code, predict the output of this program, then verify your prediction:

```
struct Box { value: any }
flux a = Box { value: [1, 2, 3] }
flux b = a
b.value.push(4)
print(len(a.value))
print(len(b.value))
```

What's Next

Structs give your data a name and a shape. But what if a value can be *one of several shapes*? A network response might be a success or a failure. A drawing command might be a circle, a rectangle, or a line. In the next chapter we meet *enums*—Lattice's answer to sum types—and discover how to model these "one of many" scenarios with precision and safety.

Chapter 16

Enums

A struct says "this value has *all* of these fields." An enum says "this value is *one* of these variants." Where structs model *product types*—a user has a name *and* an age *and* an email—enums model *sum types*: a network response is either a success *or* a failure, a shape is a circle *or* a rectangle *or* a triangle.

Lattice enums are full algebraic data types. Each variant can carry its own payload, and the compiler checks that your `match` expressions handle every variant. In this chapter we will define enums, construct variants, destructure them with pattern matching, model state machines, and build the ubiquitous `Option` and `Result` patterns.

16.1 Defining Enums With and Without Payloads

An enum definition lists a type name and a set of *variants*:

Listing 16.1: A basic enum with unit variants

```
enum Direction {
    North,
    South,
    East,
    West
}
```

Each variant is a distinct tag. A `Direction` value is exactly one of `North`, `South`, `East`, or `West`—nothing else.

Enum

An *enum* (short for "enumeration") is a type that can be one of a fixed set of *variants*. Each variant has a name and may carry a *payload*—zero or more associated values that travel with the tag.

Variants without payloads are called *unit variants*. They carry no additional data:

Listing 16.2: Unit variants carry no data

```
enum Color {
    Red,
    Green,
    Blue
}

let favourite = Color::Red
print(favourite)  // Color::Red
```

16.1.1 Variants with Payloads

A variant becomes more interesting when it carries data. List the payload types in parentheses after the variant name:

Listing 16.3: Enum with payload variants

```
enum Color {
    Red,
    Green,
    Blue,
    Rgb(any, any, any)
}

let coral = Color::Rgb(255, 127, 80)
print(coral)  // Color::Rgb(255, 127, 80)
```

The `Rgb` variant carries three values—the red, green, and blue channels. Notice that unit variants and payload variants can coexist in the same enum. You can mix and match freely.

Here is a `Shape` enum where each variant carries different amounts of data:

Listing 16.4: Variants with different payload sizes

```
enum Shape {
    Circle(any),
    Rectangle(any, any),
    Triangle(any, any, any),
    None
}
```

`Circle` takes one value (the radius), `Rectangle` takes two (width and height), `Triangle` takes three (side lengths), and `None` carries nothing.

Payload Type Annotations

In the examples above we use `any` for payload types. You can also write concrete types for documentation and strict-mode checking:

```
enum Shape {
    Circle(Float),
    Rectangle(Float, Float),
    None
}
```

This makes the intended contract clearer. Generic type parameters (e.g., `Option<T>`) are covered in Chapter 18.

16.2 Constructing Variants

To create an enum value, use the double-colon syntax: `EnumName::VariantName` for unit variants, or `EnumName::VariantName(args)` for payload variants.

Listing 16.5: Constructing enum variants

```
enum Color {
    Red,
    Green,
    Blue,
    Rgb(any, any, any)
}

let r = Color::Red
let g = Color::Green
let custom = Color::Rgb(255, 128, 0)

print(r)       // Color::Red
print(g)       // Color::Green
print(custom)  // Color::Rgb(255, 128, 0)
```

16.2.1 Enums Returned from Functions

Enums are regular values—you can pass them to functions, return them, and store them in collections:

Listing 16.6: Returning enums from functions

```
enum Color {
    Red,
    Green,
    Blue,
    Rgb(any, any, any)
}

fn make_color(kind: String) -> Color {
    if kind == "red" {
        return Color::Red
    }
    if kind == "custom" {
        return Color::Rgb(1, 2, 3)
    }
    return Color::Blue
}

print(make_color("red"))     // Color::Red
print(make_color("custom"))  // Color::Rgb(1, 2, 3)
print(make_color("other"))   // Color::Blue
```

16.2.2 Enums in Collections

Listing 16.7: An array of enum values

```
enum Status {
    Pending,
    Running(any),
    Done(any),
    Failed(any, any)
}

let tasks = [
    Status::Pending,
    Status::Running("compile"),
    Status::Done(42),
    Status::Failed("timeout", 504)
]

for task in tasks {
    print(task)
}
```

16.2.3 Runtime Introspection Methods

Every enum value carries metadata that you can inspect at runtime:

Listing 16.8: Enum introspection methods

```
enum Option {
    Some(any),
    None
}

let val = Option::Some(42)

print(val.tag())           // "Some"
print(val.enum_name())     // "Option"
print(val.variant_name())  // "Some"
print(val.payload())       // [42]
print(val.is_variant("Some"))  // true
print(val.is_variant("None"))  // false
```

The `.payload()` method returns an array of the variant's payload elements. For unit variants, it returns an empty array. These methods are useful for logging and debugging, but for most logic you should prefer pattern matching.

16.3 Matching on Enums

Pattern matching is the primary way to work with enums. If you read Chapter 9, you have already seen `match` expressions with literals, wildcards, and guards. Enum patterns add a new dimension: you can match on the variant *and* destructure the payload in a single step.

16.3.1 Matching Unit Variants

At its most basic, matching an enum looks like matching constants:

Listing 16.9: Matching unit variants

```
enum Direction {
    North,
    South,
    East,
    West
}

fn describe(dir: Direction) -> String {
    match dir {
        Direction::North => "heading north",
        Direction::South => "heading south",
        Direction::East  => "heading east",
        Direction::West  => "heading west"
    }
}

print(describe(Direction::North))  // heading north
print(describe(Direction::West))   // heading west
```

16.3.2 Destructuring Payloads

When a variant carries data, the match pattern can bind payload elements to variables:

Listing 16.10: Destructuring enum payloads

```
enum Option {
    Some(any),
    None
}

let val = Option::Some(42)

let result = match val {
    Option::Some(x) => "got: " + to_string(x),
    Option::None    => "nothing",
    _ => "unknown"
}
print(result)  // got: 42
```

The pattern `Option::Some(x)` does two things at once: it checks that the value is the `Some` variant, and it binds the payload to the variable `x`. Inside the arm's body, `x` holds the value `42`.

16.3.3 Multi-Element Payloads

Variants with multiple payload elements bind multiple variables:

Listing 16.11: Multi-binding destructuring

```
enum Pair {
    Both(any, any),
    Single(any),
    Empty
}

let p = Pair::Both("hello", 99)
let result = match p {
    Pair::Both(a, b)  => to_string(a) + "=" + to_string(b),
    Pair::Single(x)   => "single: " + to_string(x),
    Pair::Empty        => "empty",
    _ => "?"
}
print(result)  // hello=99
```

16.3.4 Using Destructured Values in Computations

The bound variables are full values—you can use them in any expression:

Listing 16.12: Computing with destructured values

```
enum Result {
    Ok(any),
    Err(any)
}

let r = Result::Ok(21)
let doubled = match r {
    Result::Ok(v)  => v * 2,
    Result::Err(e) => -1,
    _ => 0
}
print(doubled)  // 42
```

16.3.5 Block Bodies in Arms

When a single expression is not enough, use a block body:

Listing 16.13: Block bodies in match arms

```
enum Option {
    Some(any),
    None
}

let val = Option::Some(10)
let result = match val {
    Option::Some(n) => {
        let doubled = n * 2
        let tripled = n * 3
        "doubled=" + to_string(doubled) + " tripled=" + to_string(tripled)
    },
    _ => "none"
}
print(result)  // doubled=20 tripled=30
```

16.3.6 Ignoring Payload with Wildcards

If you care about the variant but not the payload, use _ in the payload position:

Listing 16.14: Ignoring payloads with wildcards

```
enum Status {
    Pending,
    Running(any),
    Done(any),
    Failed(any, any)
}

flux pending_count = 0
flux done_count = 0
let statuses = [Status::Pending, Status::Done(42), Status::Running("build")]

for s in statuses {
    match s {
        Status::Pending => { pending_count += 1 },
        Status::Done(_) => { done_count += 1 },
        _ => {}
    }
}
print(pending_count)  // 1
print(done_count)     // 1
```

16.3.7 Nested Match Expressions

You can nest match expressions to drill into layered data:

Listing 16.15: Nested match on enums

```
enum Result {
    Ok(any),
    Err(any)
}

fn classify(val: any) -> String {
    return match val {
        Result::Ok(v) => {
            match v {
                0 => "ok:zero",
                _ => "ok:" + to_string(v)
            }
        },
        Result::Err(e) => "err:" + to_string(e),
        _ => "unknown"
    }
}

print(classify(Result::Ok(0)))       // ok:zero
print(classify(Result::Ok(42)))      // ok:42
print(classify(Result::Err("bad")))  // err:bad
```

Match as an Expression

Remember that `match` is an expression in Lattice—it evaluates to a value. You can assign its result to a variable, pass it to a function, or return it directly. This makes match-based dispatch composable and concise.

16.4 Using Enums for State Machines

Enums are a natural fit for state machines because each variant represents a distinct state, and the payload carries state-specific data. Let's model a task runner that moves through a lifecycle:

Listing 16.16: A task lifecycle as an enum

```
enum TaskState {
    Pending,
    Running(any),
    Done(any),
    Failed(any, any)
}

fn status_label(s: TaskState) -> String {
    match s {
        TaskState::Pending => "pending",
        TaskState::Running(task) =>
            "running: " + to_string(task),
        TaskState::Done(result) =>
            "done: " + to_string(result),
        TaskState::Failed(err, code) =>
            "failed: " + to_string(err) + " (" + to_string(code) + ")",
        _ => "unknown"
    }
}
```

16.4.1 Driving State Transitions

A state machine advances by replacing the current state with a new variant:

Listing 16.17: State transitions

```
enum TaskState {
    Pending,
    Running(any),
    Done(any),
    Failed(any, any)
}

fn advance(state: TaskState, tick: Int) -> TaskState {
    match state {
        TaskState::Pending => TaskState::Running("job_" + to_string(tick)),
        TaskState::Running(task) => {
            if tick > 3 {
                TaskState::Done("completed " + to_string(task))
            } else {
                TaskState::Running(task)
            }
        },
        _ => state
    }
}

flux state = TaskState::Pending
for tick in 0..6 {
    state = advance(state, tick)
    print(to_string(tick) + ": " + match state {
        TaskState::Pending      => "pending",
        TaskState::Running(t)   => "running " + to_string(t),
        TaskState::Done(r)      => "done: " + to_string(r),
        TaskState::Failed(e, c) => "failed",
        _ => "?"
    })
}
```

Because the state is an enum, the compiler can warn you if you forget to handle a variant. And because each variant carries exactly the data it needs, you never have stale or irrelevant fields cluttering your state object.

16.4.2 A Parser Token Example

Enums excel at modelling token types in a parser, where each token variant carries different data:

Listing 16.18: Tokens as an enum

```
enum Token {
    Number(any),
    Identifier(any),
    Plus,
    Minus,
    Star,
    LeftParen,
    RightParen,
    Eof
}

fn token_label(tok: Token) -> String {
    match tok {
        Token::Number(n)     => "NUM(" + to_string(n) + ")",
        Token::Identifier(s) => "ID(" + to_string(s) + ")",
        Token::Plus          => "+",
        Token::Minus         => "-",
        Token::Star          => "*",
        Token::LeftParen     => "(",
        Token::RightParen    => ")",
        Token::Eof           => "EOF"
    }
}

let tokens = [
    Token::Number(42),
    Token::Plus,
    Token::Identifier("x"),
    Token::Eof
]

for tok in tokens {
    print(token_label(tok))
}
// Output: NUM(42), +, ID(x), EOF
```

16.5 Option-Like and Result-Like Patterns

Two patterns appear so frequently across programming languages that they deserve special attention: *optional values* (a value that might be absent) and *fallible operations* (a computation that might fail).

Lattice does not have built-in `Option` and `Result` types in the language itself, but you can define them as enums and they work beautifully.

16.5.1 The Option Pattern

Listing 16.19: Defining and using Option

```
enum Option {
    Some(any),
    None
}

fn find_user(id: Int) -> Option {
    if id == 1 {
        return Option::Some("Alice")
    }
    if id == 2 {
        return Option::Some("Bob")
    }
    return Option::None
}

let user = find_user(1)
match user {
    Option::Some(name) => print("Found: " + name),
    Option::None       => print("User not found"),
    _ => print("?")
}
// Output: Found: Alice
```

An `unwrap_or` helper makes optionals easier to use in expressions:

Listing 16.20: An unwrap_or helper

```
enum Option {
    Some(any),
    None
}

fn unwrap_or(opt: Option, default: any) -> any {
    match opt {
        Option::Some(v) => v,
        Option::None    => default,
        _ => default
    }
}

let a = Option::Some(10)
let b = Option::None

print(unwrap_or(a, 0))   // 10
print(unwrap_or(b, -1))  // -1
```

16.5.2 The Result Pattern

Listing 16.21: Defining and using Result

```
enum Result {
    Ok(any),
    Err(any)
}

fn parse_port(input: String) -> Result {
    let n = parse_int(input)
    if n == nil {
        return Result::Err("not a number: " + input)
    }
    if n < 1 || n > 65535 {
        return Result::Err("port out of range: " + to_string(n))
    }
    return Result::Ok(n)
}

let result = parse_port("8080")
match result {
    Result::Ok(port)  => print("Listening on port " + to_string(port)),
    Result::Err(msg)  => print("Error: " + msg),
    _ => print("?")
}
// Output: Listening on port 8080
```

16.5.3 Chaining Results

A common pattern is to chain fallible operations—if the first succeeds, feed its value into the second:

Listing 16.22: Chaining two results

```
enum Result {
    Ok(any),
    Err(any)
}

fn chain_results(a: Result, b: Result) -> Result {
    match a {
        Result::Ok(va) => {
            match b {
                Result::Ok(vb) => Result::Ok(va + vb),
                Result::Err(e) => Result::Err(e),
                _ => Result::Err("unknown")
            }
        },
        Result::Err(e) => Result::Err(e),
        _ => Result::Err("unknown")
    }
}

let r = chain_results(Result::Ok(10), Result::Ok(20))
match r {
    Result::Ok(v)  => print("ok: " + to_string(v)),
    Result::Err(e) => print("err: " + to_string(e)),
    _ => print("unknown")
}
// Output: ok: 30
```

Enums vs. try/catch

Lattice also provides `try`/`catch` for error handling (see Chapter 10). The `Result` pattern and exceptions are complementary strategies. Use `Result` when errors are *expected* and part of your data flow; use `try`/`catch` when errors are *exceptional* and you want to bail out of a computation.

16.6 Enum Exhaustiveness Guarantees

One of the most powerful features of enum-based matching is *exhaustiveness checking*. When you match on an enum, the compiler examines your arms and warns you if any variant is unhandled.

16.6.1 How It Works

The compiler (in `src/match_check.c`) performs a straightforward analysis:

1. If any arm has an unguarded wildcard (`_`) or binding pattern, the match is considered exhaustive—the catch-all handles everything.
2. Otherwise, if the patterns are enum variants, the compiler collects which variants appear and compares them against the enum's definition.
3. Any missing variants produce a warning.

Listing 16.23: An exhaustive match—no warning

```
enum Direction {
    North,
    South,
    East,
    West
}

fn label(d: Direction) -> String {
    match d {
        Direction::North => "N",
        Direction::South => "S",
        Direction::East  => "E",
        Direction::West  => "W"
    }
}
```

All four variants are covered, so the compiler is happy. Now remove one:

Listing 16.24: A non-exhaustive match—compiler warns

```
fn label(d: Direction) -> String {
    match d {
        Direction::North => "N",
        Direction::South => "S",
        Direction::East  => "E"
        // West is missing!
    }
}
// Compiler warning: non-exhaustive match: missing Direction variant `Direction::West`
```

Guarded Arms and Exhaustiveness

A guarded arm (e.g., `n if n > 0 => ...`) still counts toward variant coverage—the compiler sees that you explicitly handled that variant. However, a guarded *wildcard* does *not* count as a catch-all, because the guard might fail at runtime. If you need a true catch-all, add an unguarded `_ => ...` arm.

16.6.2 Boolean Exhaustiveness

The checker also handles boolean matches:

Listing 16.25: Boolean exhaustiveness

```
let flag = true
let label = match flag {
    true  => "yes",
    false => "no"
}
// No warning: both cases covered
```

If you omit `true` or `false`, the compiler warns about the missing case.

16.6.3 Non-Enum Matches

For integer, string, or float scrutinees, the set of possible values is unbounded. The compiler cannot enumerate them, so it requires a wildcard arm:

Listing 16.26: Wildcard required for integer match

```
let score = 85
let grade = match score {
    0..59   => "F",
    60..69  => "D",
    70..79  => "C",
    80..89  => "B",
    90..100 => "A",
    _       => "invalid"
}
print(grade)  // B
```

Without the `_ => "invalid"` arm, the compiler warns: “consider adding a wildcard _ arm.”

16.6.4 Why Exhaustiveness Matters

Exhaustiveness checking catches a common class of bugs at compile time. When you add a new variant to an enum, every `match` expression that touches that enum will produce a warning until you handle the new case. This is far safer than an `if`/`else` chain, where a missing branch silently falls through.

The "Add a Variant" Test

When designing an enum, imagine adding a new variant six months from now. Will all the match sites in your codebase light up with warnings? If you use enums and match consistently, the answer is yes—and that is a tremendous safety net for evolving codebases.

16.7 Recursive Enums

Because payload elements can be any value, enums can refer to themselves recursively. This is perfect for tree structures:

Listing 16.27: A recursive binary tree

```
enum Tree {
    Leaf(any),
    Branch(any, any)
}

fn tree_sum(t: any) -> Int {
    match t {
        Tree::Leaf(v) => v,
        Tree::Branch(l, r) => tree_sum(l) + tree_sum(r),
        _ => 0
    }
}

fn tree_depth(t: any) -> Int {
    match t {
        Tree::Leaf(_) => 1,
        Tree::Branch(l, r) => {
            let ld = tree_depth(l)
            let rd = tree_depth(r)
            if ld > rd { 1 + ld } else { 1 + rd }
        },
        _ => 0
    }
}

fn make_balanced(depth: Int, value: Int) -> Tree {
    if depth <= 0 {
        return Tree::Leaf(value)
    }
    return Tree::Branch(
        make_balanced(depth - 1, value * 2),
        make_balanced(depth - 1, value * 2 + 1)
    )
}

let tree = make_balanced(3, 1)
print("sum: " + to_string(tree_sum(tree)))
print("depth: " + to_string(tree_depth(tree)))
```

Each `Branch` holds two children that are themselves `Tree` values. The recursive functions use match to distinguish leaves from branches, and the wildcard arm handles unexpected cases gracefully.

16.7.1 An Expression Evaluator

Recursive enums can model abstract syntax trees for a mini-language:

Listing 16.28: An expression tree evaluator

```
enum Expr {
    Num(any),
    Add(any, any),
    Mul(any, any),
    Neg(any)
}

fn eval_expr(e: Expr) -> Int {
    match e {
        Expr::Num(n)    => n,
        Expr::Add(a, b) => eval_expr(a) + eval_expr(b),
        Expr::Mul(a, b) => eval_expr(a) * eval_expr(b),
        Expr::Neg(x)    => -eval_expr(x),
        _ => 0
    }
}

// Represents: (3 * 4) + (-(2))
let expr = Expr::Add(
    Expr::Mul(Expr::Num(3), Expr::Num(4)),
    Expr::Neg(Expr::Num(2))
)
print(eval_expr(expr))  // 10
```

This is a pattern you will encounter throughout compiler design, symbolic mathematics, and rule engines. Enums make the variant structure explicit; match makes the dispatch exhaustive.

16.8 Practical Patterns

16.8.1 Enums and Loops

Building enum values inside loops and processing them with match is a common workflow:

Listing 16.29: Batch-constructing enums

```
enum Status {
    Pending,
    Running(any),
    Done(any),
    Failed(any, any)
}

flux statuses = []
for i in 0..8 {
    let s = match i % 4 {
        0 => Status::Pending,
        1 => Status::Running("task_" + to_string(i)),
        2 => Status::Done(i * 10),
        _ => Status::Failed("err_" + to_string(i), i)
    }
    statuses.push(s)
}

flux pending_count = 0
flux done_count = 0
for s in statuses {
    match s {
        Status::Pending => { pending_count += 1 },
        Status::Done(_) => { done_count += 1 },
        _ => {}
    }
}
print("pending: " + to_string(pending_count))  // pending: 2
print("done: " + to_string(done_count))        // done: 2
```

16.8.2 Enums as Function Return Types

Use enums to signal outcomes without exceptions:

Listing 16.30: Validation with enums

```
enum ValidationResult {
    Valid,
    TooShort(any),
    TooLong(any),
    InvalidChar(any)
}

fn validate_username(name: String) -> ValidationResult {
    if len(name) < 3 {
        return ValidationResult::TooShort(len(name))
    }
    if len(name) > 20 {
        return ValidationResult::TooLong(len(name))
    }
    return ValidationResult::Valid
}

let result = validate_username("Al")
match result {
    ValidationResult::Valid         => print("Username accepted"),
    ValidationResult::TooShort(n)  => print("Too short: " + to_string(n) + " chars"),
    ValidationResult::TooLong(n)   => print("Too long: " + to_string(n) + " chars"),
    ValidationResult::InvalidChar(c) => print("Bad char: " + to_string(c)),
    _ => print("?")
}
// Output: Too short: 2 chars
```

16.8.3 Under the Hood

At runtime, an enum value is represented as a tagged union (see `include/value.h`):

- **`enum_name`**: the enum's type name (e.g., "Option")
- **`variant_name`**: the variant (e.g., "Some")
- **`payload`**: an array of values (empty for unit variants)
- **`payload_count`**: the number of payload elements

The compiler emits `OP_BUILD_ENUM` with the enum name, variant name, and payload count as operands. The VM pops payload values from the stack and assembles the enum value. You can explore the compilation logic in `src/stackcompiler.c`—look for the `EXPR_ENUM_VARIANT` case.

16.9 Exercises

1. **Traffic light.** Define an enum `Light` with variants `Red`, `Yellow`, and `Green`. Write a function `next_light(l: Light) -> Light` that cycles through the states: Green → Yellow → Red → Green. Run the cycle 6 times and print each state.

2. **Shape areas.** Define an enum `Shape` with variants `Circle(any)`, `Rectangle(any, any)`, and `Square(any)`. Write a function that returns the area. Use 3.14159 for pi.

3. **Linked list.** Model a linked list with an enum: `enum List { Cons(any, any), Nil }`. Write functions `list_sum` (sum all elements) and `list_len` (count elements). Build a list of `[1, 2, 3, 4, 5]` and verify both functions.

4. **Result pipeline.** Write a function `map_result(r: Result, f: Fn) -> Result` that applies `f` to the value inside `Result::Ok` and passes `Result::Err` through unchanged. Chain three `map_result` calls to build a pipeline.

5. **Exhaustiveness experiment.** Define an enum with five variants. Write a match expression that covers only three of them (no wildcard). Compile the program and observe the compiler warning. Then fix it by adding the missing variants.

What's Next

We have seen how to group fields into structs and model alternatives with enums. But so far, attaching methods to a struct has required declaring a trait—even when the trait is implemented by only one type. In the next chapter we explore *traits and impl blocks* in depth: how to declare interfaces, implement them for multiple types, and use trait-driven polymorphism to write code that works across any struct that speaks the right protocol.

Chapter 17

Traits and Impl Blocks

In Chapter 15 we attached behaviour to structs through traits and impl blocks, but we treated them as a recipe to follow without explaining the ingredients. This chapter puts traits centre stage. We will see how to declare interfaces, implement them for multiple types, compose impl blocks, and use traits as the backbone of polymorphic design.

Think of a trait as a *contract*—a promise that a type will provide certain methods. An impl block is the *fulfilment* of that promise. By separating the "what" (the trait) from the "how" (the impl), Lattice lets you write code that operates on *any type that speaks a given protocol*, without knowing the concrete type in advance.

17.1 Declaring Traits

A trait declaration lists method signatures with no bodies:

Listing 17.1: A trait with a single method

```
trait Describable {
    fn describe(self: any) -> String
}
```

The keyword `trait` is followed by a name (conventionally an adjective or capability noun), then braces containing one or more method signatures. Each signature specifies:

- The method name.

- A parameter list. The first parameter is almost always `self: any`, representing the value the method is called on.
- An optional return type.

Trait

A *trait* is a named set of method signatures that defines a behavioural interface. It contains no implementation—only declarations of the methods a type must provide.

17.1.1 Multi-Method Traits

A trait can declare any number of methods:

Listing 17.2: A trait with multiple methods

```
trait Collection {
    fn push_item(self: any, item: any) -> any
    fn pop_item(self: any) -> any
    fn peek_item(self: any) -> any
    fn is_empty(self: any) -> any
}
```

This `Collection` trait declares four methods. Any type that implements `Collection` is expected to provide all four.

17.1.2 Traits with Extra Parameters

Methods are not limited to `self`. They can accept additional parameters:

Listing 17.3: Traits with extra parameters

```
trait Scalable {
    fn scale(self: any, factor: any) -> any
}

trait Combinable {
    fn combine(self: any, other: any) -> any
}
```

The `Scalable` trait expects a method that takes a scaling factor; `Combinable` expects a method that merges two values of the same kind.

Trait Bodies Are Signatures Only

You cannot write method bodies inside a trait declaration. The trait is purely a contract. All implementation goes in `impl` blocks. This is a deliberate design choice: it keeps the interface description separate from the implementation, making both easier to read.

17.2 Implementing Traits for Structs

An `impl` block provides method bodies for a specific type:

Listing 17.4: Implementing a trait

```
struct Animal { kind: any, name: any, legs: any }

trait Displayable {
    fn display(self: any) -> String
}

impl Displayable for Animal {
    fn display(self: any) -> String {
        return self.name + " the " + self.kind
            + " (" + to_string(self.legs) + " legs)"
    }
}

let dog = Animal { kind: "dog", name: "Rex", legs: 4 }
print(dog.display())  // Rex the dog (4 legs)
```

The syntax is `impl TraitName for TypeName { ... }`, and inside the braces you write full function definitions. Each method must match a signature declared in the trait.

17.2.1 How `self` Works

The first parameter, `self`, receives a *copy* of the value the method was called on. This follows Lattice's value semantics:

Listing 17.5: Self is a copy

```
struct Counter { value: any }

trait Incrementable {
    fn increment(self: any) -> Counter
}

impl Incrementable for Counter {
    fn increment(self: any) -> Counter {
        return Counter { value: self.value + 1 }
    }
}

let c = Counter { value: 10 }
let c2 = c.increment()
print(c.value)   // 10 (unchanged)
print(c2.value)  // 11
```

Because `self` is a copy, "mutating" methods must return a new instance. The original remains untouched. This is the same pattern we saw with struct methods in Section 15.3.

17.2.2 Accessing Fields through `self`

Inside a method body, `self` is a regular struct value. You read its fields with dot notation:

Listing 17.6: Reading fields from self

```
struct Rect { w: any, h: any }

trait Geometry {
    fn area(self: any) -> any
    fn perimeter(self: any) -> any
}

impl Geometry for Rect {
    fn area(self: any) -> any {
        return self.w * self.h
    }

    fn perimeter(self: any) -> any {
        return 2 * (self.w + self.h)
    }
}

let r = Rect { w: 5, h: 3 }
print(r.area())       // 15
print(r.perimeter())  // 16
```

17.2.3 Method Chaining

When a method returns a struct of the same type, you can chain calls:

Listing 17.7: Method chaining

```
struct Stack { items: any, size: any }

trait Collection {
    fn push_item(self: any, item: any) -> any
    fn peek_item(self: any) -> any
}

impl Collection for Stack {
    fn push_item(self: any, item: any) -> any {
        flux new_items = []
        for i in 0..self.size {
            new_items.push(self.items[i])
        }
        new_items.push(item)
        return Stack { items: new_items, size: self.size + 1 }
    }

    fn peek_item(self: any) -> any {
        if self.size == 0 {
            return nil
        }
        return self.items[self.size - 1]
    }
}

let s = Stack { items: [], size: 0 }
let top = s.push_item(1).push_item(2).push_item(3).peek_item()
print(top)  // 3
```

Each `push_item` returns a new `Stack`, so we can immediately call the next method on the result. The chain reads like a pipeline: push 1, push 2, push 3, peek.

Chaining Reads Naturally

Method chaining works because Lattice methods that return the same type create a "fluent interface." This is a powerful pattern for building up data structures or transformation pipelines without intermediate variables.

17.3 Multiple Impl Blocks per Struct

A struct can implement as many traits as it needs. Each trait gets its own `impl` block:

Listing 17.8: Three traits on one struct

```
struct Point { x: any, y: any }

trait Describable {
    fn describe(self: any) -> String
}

trait HasArea {
    fn area(self: any) -> any
}

trait Scalable {
    fn scale(self: any, factor: any) -> any
}

impl Describable for Point {
    fn describe(self: any) -> String {
        return "Point(" + to_string(self.x) + ", " + to_string(self.y) + ")"
    }
}

impl HasArea for Point {
    fn area(self: any) -> any {
        return self.x * self.y
    }
}

impl Scalable for Point {
    fn scale(self: any, factor: any) -> any {
        return Point { x: self.x * factor, y: self.y * factor }
    }
}

let p = Point { x: 3, y: 4 }
print(p.describe())  // Point(3, 4)
print(p.area())      // 12

let q = p.scale(3)
print(q.x)  // 9
print(q.y)  // 12
```

The `Point` struct now speaks three protocols: `Describable`, `HasArea`, and `Scalable`. Each impl block is self-contained and can be placed anywhere in the same file (or module).

17.3.1 How Method Registration Works

Under the hood, the compiler registers each method as a global function with a composite key. When you write:

```
impl HasArea for Rect {
    fn area(self: any) -> any { ... }
}
```

the compiler emits the method body as a function and stores it under the key `Rect::area`. When you later call `r.area()`, the VM:

1. Checks for a callable field named `area` on the struct (there is none).
2. Looks up the global `Rect::area`.
3. Calls it with a clone of `r` as `self`.

You can see this in `src/stackcompiler.c`: the `ITEM_IMPL` case builds the key string `"TypeName::methodName"` and emits `OP_DEFINE_GLOBAL` with that key. The VM's `OP_INVOKE` handler resolves the method at call time.

Callable Fields Take Priority

If a struct has both a closure field and a trait method with the same name, the closure field wins. The runtime checks fields first, then falls through to the impl registry. This is by design: it lets you override a trait method on a per-instance basis if needed (see Section 15.4).

17.4 Designing with Interfaces

Traits are Lattice's interface mechanism. Effective use of traits leads to code that is modular, testable, and extensible. Here are some patterns.

17.4.1 The Same Trait on Different Types

The real power of traits emerges when multiple types implement the same interface. Functions that operate on the trait—rather than a specific struct—work with *any* implementing type:

Listing 17.9: One trait, two types

```
struct Circle { radius: Float }
struct Rect { width: Float, height: Float }

trait Describable {
    fn describe(self: any) -> String
}

trait HasArea {
    fn area(self: any) -> Float
}

impl Describable for Circle {
    fn describe(self: any) -> String {
        return "Circle(r=" + to_string(self.radius) + ")"
    }
}

impl HasArea for Circle {
    fn area(self: any) -> Float {
        return 3.14159 * self.radius * self.radius
    }
}

impl Describable for Rect {
    fn describe(self: any) -> String {
        return "Rect(" + to_string(self.width) + "x" + to_string(self.height) + ")"
    }
}

impl HasArea for Rect {
    fn area(self: any) -> Float {
        return self.width * self.height
    }
}

let c = Circle { radius: 5.0 }
let r = Rect { width: 4.0, height: 6.0 }

print(c.describe())  // Circle(r=5)
print(c.area())      // 78.53975

print(r.describe())  // Rect(4x6)
print(r.area())      // 24
```

Both `Circle` and `Rect` implement `Describable` and `HasArea`. The methods have different bodies—each type knows how to describe and measure itself—but the *interface* is identical.

17.4.2 Traits for Separation of Concerns

Suppose you are building a notification system. You might define separate traits for each concern:

Listing 17.10: Separating concerns with traits

```
struct EmailAlert {
    to: String,
    subject: String,
    body: String
}

trait Sendable {
    fn send(self: any) -> String
}

trait Loggable {
    fn log_entry(self: any) -> String
}

impl Sendable for EmailAlert {
    fn send(self: any) -> String {
        return "Sending email to " + self.to + ": " + self.subject
    }
}

impl Loggable for EmailAlert {
    fn log_entry(self: any) -> String {
        return "[EMAIL] to=" + self.to + " subject=" + self.subject
    }
}

let alert = EmailAlert {
    to: "ops@example.com",
    subject: "Server down",
    body: "The production server is unreachable."
}

print(alert.send())      // Sending email to ops@example.com: Server down
print(alert.log_entry()) // [EMAIL] to=ops@example.com subject=Server down
```

Each trait captures one aspect of behaviour. A `SlackAlert` struct could implement the same traits with different bodies, and any code that calls `.send()` or `.log_entry()` would work with either type.

17.4.3 Naming Conventions

Good trait names describe a capability, not a thing:

- `Describable`, `Displayable` — "can be described / displayed"
- `HasArea`, `HasLength` — "possesses a measurable property"
- `Scalable`, `Combinable` — "can be scaled / combined"
- `Collection`, `Geometry` — group of related operations

When you read `impl Scalable for Vec2`, the intent is immediately clear: "`Vec2` can be scaled."

17.5 Trait-Driven Polymorphism

Because Lattice is dynamically typed, any value can be passed to any function. Traits add a *semantic* layer on top: they document which methods a value must support, and the runtime dispatches to the correct implementation based on the value's type.

17.5.1 Heterogeneous Collections

You can put different types in the same array and call the same method on each:

Listing 17.11: Polymorphic dispatch over an array

```
struct Circle { radius: Float }
struct Rect { width: Float, height: Float }

trait HasArea {
    fn area(self: any) -> Float
}

impl HasArea for Circle {
    fn area(self: any) -> Float {
        return 3.14159 * self.radius * self.radius
    }
}

impl HasArea for Rect {
    fn area(self: any) -> Float {
        return self.width * self.height
    }
}

let shapes = [
    Circle { radius: 3.0 },
    Rect { width: 4.0, height: 5.0 },
    Circle { radius: 1.0 }
]

flux total_area = 0.0
for shape in shapes {
    total_area = total_area + shape.area()
}
print(total_area)  // 28.27431 + 20.0 + 3.14159 = 51.4159
```

The `for` loop does not know whether each element is a `Circle` or a `Rect`. It calls `.area()` on each, and the runtime dispatches to the correct implementation based on the struct's type name.

Dynamic Dispatch

Lattice uses *dynamic dispatch* for trait methods. When you call `shape.area()`, the VM looks up the method key `TypeName::area` at runtime, using the actual type of the value. This is different from static dispatch (resolved at compile time) used by languages like Rust.

17.5.2 Methods in Expressions and Conditionals

Trait method calls are expressions—you can use them in `if` conditions, arithmetic, and anywhere else a value is expected:

Listing 17.12: Methods in expressions

```
struct Rect { w: any, h: any }

trait Geometry {
    fn area(self: any) -> any
}

impl Geometry for Rect {
    fn area(self: any) -> any {
        return self.w * self.h
    }
}

let r1 = Rect { w: 4, h: 6 }
let r2 = Rect { w: 3, h: 8 }

let total_area = r1.area() + r2.area()
print(total_area)  // 48

let larger = if r1.area() > r2.area() { "r1" } else { "r2" }
print(larger)  // r2
```

17.5.3 Methods in Loops

A common pattern is to accumulate results by calling trait methods inside a loop:

Listing 17.13: Accumulating over trait methods

```
struct Rect { w: any, h: any }

trait Geometry {
    fn area(self: any) -> any
}

impl Geometry for Rect {
    fn area(self: any) -> any {
        return self.w * self.h
    }
}

let rects = [
    Rect { w: 1, h: 1 },
    Rect { w: 2, h: 2 },
    Rect { w: 3, h: 3 }
]

flux total = 0
for r in rects {
    total = total + r.area()
}
print(total)  // 1 + 4 + 9 = 14
```

17.5.4 Traits on Enums

Traits are not limited to structs—you can implement them for enums too:

Listing 17.14: A trait implemented for an enum

```
enum Status {
    Active,
    Inactive,
    Error(any)
}

trait Displayable {
    fn display(self: any) -> String
}

impl Displayable for Status {
    fn display(self: any) -> String {
        return "Status:active"
    }
}

let s = Status::Active
print(s.display())  // Status:active
```

Enum Method Dispatch

When you call a trait method on an enum value, the runtime looks up the method key using the *enum type name*—not the variant name. So `Status::Active`, `Status::Inactive`, and `Status::Error("oops")` all dispatch to the same impl block. If you need per-variant behaviour, use `match` inside the method body.

17.5.5 A Complete Polymorphism Example

Let's put it all together with a two-dimensional vector type that supports scaling and combining:

Listing 17.15: A complete trait-driven design

```
struct Vec2 { x: any, y: any }

trait Describable {
    fn describe(self: any) -> String
}

trait Scalable {
    fn scale(self: any, factor: any) -> any
}

trait Combinable {
    fn combine(self: any, other: any) -> any
}

impl Describable for Vec2 {
    fn describe(self: any) -> String {
        return "Vec2(" + to_string(self.x) + ", " + to_string(self.y) + ")"
    }
}

impl Scalable for Vec2 {
    fn scale(self: any, factor: any) -> any {
        return Vec2 { x: self.x * factor, y: self.y * factor }
    }
}

impl Combinable for Vec2 {
    fn combine(self: any, other: any) -> any {
        return Vec2 { x: self.x + other.x, y: self.y + other.y }
    }
}

let velocity = Vec2 { x: 3, y: 4 }
let boost = velocity.scale(2)
print(boost.describe())  // Vec2(6, 8)

let gravity = Vec2 { x: 0, y: -1 }
let result = boost.combine(gravity)
print(result.describe())  // Vec2(6, 7)
```

This design is extensible. If you later add a `Vec3` struct, you can implement the same three traits for it, and any code that operates on `Describable`, `Scalable`, or `Combinable` values will work with both types—no changes needed.

17.6 Under the Hood

Understanding the implementation helps when debugging method dispatch or designing complex trait hierarchies.

17.6.1 Compilation

Traits themselves are metadata-only. The compiler sees a trait declaration and records it, but emits no bytecode for it. Only `impl` blocks produce code:

1. Each method body is compiled as a regular function.
2. The compiled function is registered as a global under the key `"TypeName::methodName"`.
3. The VM resolves this key at call time via the `OP_INVOKE` opcode family.

You can see this in `src/stackcompiler.c`—the `ITEM_TRAIT` case is essentially a no-op, while `ITEM_IMPL` compiles each method and emits `OP_DEFINE_GLOBAL`.

17.6.2 Dispatch Performance

The stack VM uses three specialised invoke opcodes for performance:

- **`OP_INVOKE_LOCAL`**: When the receiver is a local variable, the VM can skip the stack and access the value directly from the frame's slot array. This is the fastest path.
- **`OP_INVOKE_GLOBAL`**: When the receiver is a global variable. Slightly slower due to the global lookup.
- **`OP_INVOKE`**: The general case for arbitrary expressions. The struct is on the stack, and the VM searches for the method.

Each path also maintains a *polymorphic inline cache* (PIC) to avoid repeated lookups when the same call site sees the same type across iterations. You can explore this in `src/stackvm.c`.

17.6.3 Self Cloning

When a method is invoked, the runtime clones `self` via `value_deep_clone()` before passing it to the method frame. This ensures that mutations inside the method cannot accidentally affect the caller's copy—maintaining Lattice's value semantics guarantee.

17.7 Exercises

1. **Shape hierarchy.** Define structs `Circle`, `Square`, and `Triangle`. Create a `HasArea` trait and implement it for all three. Put instances of each into an array and compute the total area in a loop.

2. **Stack from scratch.** Using the `Collection` trait from this chapter (with methods `push_item`, `pop_item`, `peek_item`, and `is_empty`), implement a `Queue` struct that supports FIFO ordering. Verify that items come out in insertion order.

3. **Composing traits.** Define two traits: `Serializable` (with a `to_string_repr` method) and `Parseable` (with a class-level `from_string` function). Implement both for a `Temperature` struct that holds a value and a unit (`"C"` or `"F"`). Demonstrate round-tripping: convert to a string and parse back.

4. **Trait method chaining.** Create a `Builder` struct with fields `parts` and `count`. Implement a `Buildable` trait with methods `add_part` (returns a new Builder with the part appended) and `build` (returns a summary string). Chain at least four `add_part` calls followed by `build`.

What's Next

Traits let you write code that works across multiple types. But so far, every type annotation in our examples has been `any`—a catch-all that says nothing about what the value actually is. In the next chapter we meet *generics*, which let you parameterise functions, structs, and enums over types. Instead of `fn identity(x: any) -> any`, you will write `fn identity<T>(x: T) -> T`—and the language will understand the relationship between input and output.

Chapter 18

Generics

Up to this point, every type annotation in our structs, enums, and functions has been either a concrete type like `Int` or the catch-all `any`. Concrete types are precise but inflexible; `any` is flexible but tells the reader nothing about intent. Generics occupy the middle ground: they let you write code that is *parameterised* over types, so that the shape of the data is documented even when the specific type is left open.

In Lattice, generics are a *documentation and design* mechanism. The language parses generic type parameters on functions, structs, and enums, and it supports rich type expressions like `Map<String, Int>` and `Fn(Int) -> Bool`. At runtime, however, Lattice is dynamically typed—type parameters are erased and values flow freely. This chapter shows you how to use generics to write clearer, more intention-revealing code.

18.1 Generic Functions

A generic function declares one or more type parameters in angle brackets between the function name and the parameter list:

Listing 18.1: A generic identity function

```
fn identity<T>(x: T) -> T {
    return x
}

print(identity(42))       // 42
print(identity("hello"))  // hello
print(identity(true))     // true
```

The <T> after the function name introduces a type parameter named T. We then use T in the parameter type and the return type to express that "whatever type goes in, the same type comes out."

Type Parameter

A *type parameter* is a placeholder name (conventionally a single uppercase letter like T, U, or K) that stands for a type to be specified later. It appears between angle brackets in a declaration: `fn foo<T>(x: T) -> T`.

18.1.1 Multiple Type Parameters

Functions can accept more than one type parameter:

Listing 18.2: Multiple type parameters

```
fn pair<A, B>(first: A, second: B) -> any {
    return [first, second]
}

let result = pair(1, "two")
print(result)  // [1, two]
```

The names A and B signal to the reader that the first and second arguments may be different types.

18.1.2 When to Use Generic Functions

Generic type parameters shine when a function's logic is independent of the specific type. Classic examples include:

Listing 18.3: Generic utility functions

```
fn first_of<T>(items: [T]) -> T {
    return items[0]
}

fn swap<A, B>(a: A, b: B) -> any {
    return [b, a]
}

fn repeat<T>(value: T, count: Int) -> [T] {
    flux result = []
    for i in 0..count {
        result.push(value)
    }
    return result
}

print(first_of([10, 20, 30]))    // 10
print(swap("hello", 42))         // [42, hello]
print(repeat("ha", 3))           // [ha, ha, ha]
```

Without generics, these functions would use `any` for every parameter—which is *correct* but says nothing about the relationship between inputs and outputs. The generic signatures communicate intent: `first_of<T>` takes an array of `T` and returns a `T`, so you know the return type matches the element type.

Generics Are Erased at Runtime

Lattice parses generic type parameters and stores them in the AST, but they are *not enforced* at runtime. The runtime treats all values as dynamically typed. This means generics serve as *documentation* and *design intent*—they guide human readers and future tooling, but the compiler does not yet reject type mismatches. Think of them as structured comments with syntactic validation.

18.1.3 Generic Functions with Constraints

Although Lattice does not enforce type constraints today, you can document expected capabilities in comments:

Listing 18.4: Documenting type expectations

```
// T must support the `+` operator
fn sum_all<T>(items: [T], zero: T) -> T {
    flux total = zero
    for item in items {
        total = total + item
    }
    return total
}

print(sum_all([1, 2, 3, 4], 0))               // 10
print(sum_all([1.5, 2.5, 3.0], 0.0))          // 7.0
print(sum_all(["a", "b", "c"], ""))            // abc
```

The function works for any type that supports addition. The generic parameter `T` communicates that the accumulator, the elements, and the result are all the same type.

18.2 Generic Structs

Structs can also declare type parameters:

Listing 18.5: A generic Box struct

```
struct Box<T> {
    value: T
}

let int_box = Box { value: 42 }
let str_box = Box { value: "hello" }

print(int_box.value)  // 42
print(str_box.value)  // hello
```

The `<T>` after the struct name introduces a type parameter. The field `value` is annotated with `T`, signalling that a `Box` holds exactly one value of a uniform type.

Instantiation Does Not Require Type Arguments

When you write `Box { value: 42 }`, you do not need to specify `Box<Int> { value: 42 }`. The type parameter exists in the *definition* to document the struct's generic nature; at construction time, the value itself determines the concrete type.

18.2.1 Multi-Parameter Structs

Listing 18.6: A struct with two type parameters

```
struct Pair<A, B> {
    first: A,
    second: B
}

let entry = Pair { first: "name", second: 42 }
print(entry.first)   // name
print(entry.second)  // 42
```

The `Pair<A, B>` struct makes it clear that the two fields may hold different types. Contrast this with:

```
struct Pair { first: any, second: any }
```

Both definitions compile identically, but the generic version tells the reader "these are distinct type roles, not arbitrary values."

18.2.2 Realistic Generic Structs

Let's build a generic key–value entry and a lightweight container:

Listing 18.7: Generic key-value entry

```
struct Entry<K, V> {
    key: K,
    value: V
}

fn make_entry<K, V>(key: K, value: V) -> Entry {
    return Entry { key: key, value: value }
}

let user_entry = make_entry("alice", 30)
print(user_entry.key)    // alice
print(user_entry.value)  // 30
```

Listing 18.8: A generic wrapper with a label

```
struct Tagged<T> {
    tag: String,
    data: T
}

let measurement = Tagged { tag: "temperature", data: 23.5 }
let flag = Tagged { tag: "active", data: true }

print(measurement.tag + ": " + to_string(measurement.data))  // temperature: 23.5
print(flag.tag + ": " + to_string(flag.data))                // active: true
```

18.2.3 Generic Structs with Traits

Generic structs work seamlessly with trait implementations:

Listing 18.9: Traits on generic structs

```
struct Box<T> {
    value: T
}

trait Describable {
    fn describe(self: any) -> String
}

impl Describable for Box {
    fn describe(self: any) -> String {
        return "Box(" + to_string(self.value) + ")"
    }
}

let b = Box { value: 42 }
print(b.describe())  // Box(42)

let s = Box { value: "world" }
print(s.describe())  // Box(world)
```

The impl block says `impl Describable for Box` (without type parameters)—because at runtime, all `Box` instances share the same type name regardless of what `T` was.

18.3 Generic Enums and Trait Implementations

Enums are perhaps the most natural fit for generics. The `Option<T>` and `Result<T, E>` patterns from Chapter 16 become far more expressive with type parameters:

Listing 18.10: A generic Option enum

```
enum Option<T> {
    Some(T),
    None
}

let found = Option::Some(42)
let missing = Option::None

print(found)    // Option::Some(42)
print(missing)  // Option::None
```

The type parameter `<T>` makes the intent unmistakable: `Some` carries a value of type `T`, and `None` carries nothing.

18.3.1 Generic Result

Listing 18.11: A generic Result enum

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}

fn divide(a: Int, b: Int) -> Result {
    if b == 0 {
        return Result::Err("division by zero")
    }
    return Result::Ok(a / b)
}

let r = divide(10, 3)
match r {
    Result::Ok(v)  => print("result: " + to_string(v)),
    Result::Err(e) => print("error: " + e),
    _ => print("?")
}
// Output: result: 3
```

With `Result<T, E>`, the definition documents that the success payload is one type and the error payload is another. The two-parameter signature mirrors the Rust and Haskell `Either` type.

18.3.2 Recursive Generic Enums

The `Tree` example from Section 16.7 becomes more self-documenting with a type parameter:

Listing 18.12: A generic binary tree

```
enum Tree<T> {
    Leaf(T),
    Branch(any, any)
}

fn tree_sum(t: any) -> Int {
    match t {
        Tree::Leaf(v)      => v,
        Tree::Branch(l, r) => tree_sum(l) + tree_sum(r),
        _ => 0
    }
}

let tree = Tree::Branch(
    Tree::Branch(Tree::Leaf(1), Tree::Leaf(2)),
    Tree::Leaf(3)
)
print(tree_sum(tree))  // 6
```

Branch Payloads and Self-Reference

You might notice that `Branch` uses `any` for its children rather than `Tree<T>`. This is because Lattice's type system does not yet resolve recursive type references within enum payloads. Using `any` for the recursive positions is the idiomatic approach until the type system evolves.

18.4 Type Expressions

Beyond simple names like `Int` and `String`, Lattice supports a rich syntax for *type expressions*—the annotations you write after colons in parameter lists and struct fields.

18.4.1 Named Types

The most basic type expression is a single name:

```
fn greet(name: String) -> String {
    return "Hello, " + name
}
```

Built-in named types include `Int`, `Float`, `Bool`, `String`, `Array`, `Map`, `Set`, `Tuple`, `Channel`, `Buffer`, `Iterator`, `Ref`, and `Fn`. Custom struct and enum names are also valid.

18.4.2 Parameterised Named Types

Named types can take type arguments in angle brackets:

Listing 18.13: Parameterised type expressions

```
struct Config {
    headers: Map<String, String>,
    ports: Array<Int>,
    handler: Fn
}
```

`Map<String, String>` says "a map from strings to strings." `Array<Int>` says "an array of integers." The parser recognises the angle brackets and stores the type arguments in the AST (see `src/parser.c`, the `parse_type_expr` function).

18.4.3 Array Types

Lattice supports a bracket syntax for array types:

Listing 18.14: Array type annotation

```
fn sum(nums: [Int]) -> Int {
    flux total = 0
    for n in nums {
        total = total + n
    }
    return total
}

print(sum([1, 2, 3, 4, 5]))  // 15
```

The annotation `[Int]` is shorthand for "an array whose elements are integers." It is syntactically distinct from `Array<Int>` but carries the same meaning.

18.4.4 Function Types

Function types describe the signature of a callable value:

Listing 18.15: Function type annotations

```
fn apply(f: Fn(Int) -> Int, x: Int) -> Int {
    return f(x)
}

let double = |n| { return n * 2 }
print(apply(double, 5))  // 10
```

The type `Fn(Int) -> Int` means "a function that takes an `Int` and returns an `Int`." Multi-parameter function types list all parameter types:

Listing 18.16: Multi-parameter function types

```
fn apply_binary(f: Fn(Int, Int) -> Int, a: Int, b: Int) -> Int {
    return f(a, b)
}

let add = |x, y| { return x + y }
print(apply_binary(add, 3, 4))  // 7
```

18.4.5 Combining Type Expressions

These forms compose naturally. Here is a function that takes a transformation function and an array, and returns a new array:

Listing 18.17: Combining function and array types

```
fn transform<T, U>(items: [T], f: Fn(T) -> U) -> [U] {
    flux result = []
    for item in items {
        result.push(f(item))
    }
    return result
}

let names = ["alice", "bob", "charlie"]
let lengths = transform(names, |s| { return len(s) })
print(lengths)  // [5, 3, 7]
```

The signature `fn transform<T, U>(items: [T], f: Fn(T) -> U) -> [U]` is dense but readable: it takes an array of `T`, a function from `T` to `U`, and returns an array of `U`. This is the classic "map" operation expressed with full type information.

Type Expressions as Documentation

Even though Lattice does not enforce type expressions at runtime, they serve as *machine-readable documentation*. Future tooling—such as the language server, linter, and doc generator—can leverage these annotations to provide better auto-complete, hover information, and static analysis. Write type expressions for your future self and your teammates.

18.4.6 Phase Annotations in Types

Type expressions can also include phase qualifiers from the phase system:

Listing 18.18: Phase-annotated types

```
fn freeze_point(p: ~Point) -> *Point {
    return freeze p
}
```

The tilde (~) denotes a fluid (mutable) value; the asterisk (*) denotes a crystal (frozen) value. These can be combined with any type expression, including generic ones. Phase annotations are covered in depth in Chapter 11.

18.5 Practical Patterns with Generics

Let's explore several design patterns that benefit from generic annotations.

18.5.1 The Wrapper Pattern

A common need is to wrap a value with metadata:

Listing 18.19: A timestamped wrapper

```
struct Timestamped<T> {
    value: T,
    created_at: Int
}

fn stamp<T>(value: T, time: Int) -> Timestamped {
    return Timestamped { value: value, created_at: time }
}

let record = stamp("measurement: 23.5C", 1700000000)
print(record.value)       // measurement: 23.5C
print(record.created_at)  // 1700000000
```

The type parameter `T` signals that `Timestamped` is agnostic about what it wraps. It could hold a string, an integer, or another struct.

18.5.2 The Builder Pattern

Generic structs pair well with method chaining:

Listing 18.20: A generic builder

```
struct Builder<T> {
    parts: [T],
    count: Int
}

trait Buildable {
    fn add(self: any, item: any) -> any
    fn items(self: any) -> any
}

impl Buildable for Builder {
    fn add(self: any, item: any) -> any {
        flux new_parts = []
        for i in 0..self.count {
            new_parts.push(self.parts[i])
        }
        new_parts.push(item)
        return Builder { parts: new_parts, count: self.count + 1 }
    }

    fn items(self: any) -> any {
        return self.parts
    }
}

let list = Builder { parts: [], count: 0 }
let result = list.add("alpha").add("beta").add("gamma")
print(result.items())  // [alpha, beta, gamma]
```

18.5.3 Higher-Order Generic Functions

Generic annotations make higher-order functions self-documenting:

Listing 18.21: Generic filter and reduce

```
fn filter<T>(items: [T], predicate: Fn(T) -> Bool) -> [T] {
    flux result = []
    for item in items {
        if predicate(item) {
            result.push(item)
        }
    }
    return result
}

fn reduce<T, U>(items: [T], initial: U, combine: Fn(U, T) -> U) -> U {
    flux acc = initial
    for item in items {
        acc = combine(acc, item)
    }
    return acc
}

let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

let evens = filter(nums, |n| { return n % 2 == 0 })
print(evens)  // [2, 4, 6, 8, 10]

let total = reduce(nums, 0, |acc, n| { return acc + n })
print(total)  // 55
```

The signature of `reduce<T, U>` makes the roles crystal clear: `T` is the element type, `U` is the accumulator type, and the `combine` function bridges the two.

18.5.4 Generic Option Utilities

Building on the `Option` enum from Section 16.5:

Listing 18.22: Generic Option helpers

```
enum Option<T> {
    Some(T),
    None
}

fn map_option<T, U>(opt: Option, f: Fn(T) -> U) -> Option {
    match opt {
        Option::Some(v) => Option::Some(f(v)),
        Option::None    => Option::None,
        _ => Option::None
    }
}

fn unwrap_or<T>(opt: Option, default: T) -> T {
    match opt {
        Option::Some(v) => v,
        Option::None    => default,
        _ => default
    }
}

let name = Option::Some("Alice")
let upper = map_option(name, |s| { return s.to_upper() })
print(unwrap_or(upper, "unknown"))  // ALICE

let missing = Option::None
print(unwrap_or(missing, "anonymous"))  // anonymous
```

These utility functions work with any `Option`, regardless of the payload type, because the logic depends only on the variant, not on `T`.

18.5.5 Combining Generics, Enums, and Traits

Here is a pattern that brings all three features together—a generic "result handler" that describes and processes outcomes:

Listing 18.23: Combining generics with enums and traits

```
enum Outcome<T> {
    Success(T),
    Failure(any)
}

struct Report<T> {
    outcome: Outcome,
    label: String
}

trait Summarizable {
    fn summary(self: any) -> String
}

impl Summarizable for Report {
    fn summary(self: any) -> String {
        let status = match self.outcome {
            Outcome::Success(v) => "OK: " + to_string(v),
            Outcome::Failure(e) => "FAIL: " + to_string(e),
            _ => "UNKNOWN"
        }
        return "[" + self.label + "] " + status
    }
}

let good = Report {
    outcome: Outcome::Success(200),
    label: "health check"
}

let bad = Report {
    outcome: Outcome::Failure("timeout"),
    label: "database ping"
}

print(good.summary())  // [health check] OK: 200
print(bad.summary())   // [database ping] FAIL: timeout
```

The `Report<T>` struct holds an `Outcome<T>` enum, and the `Summarizable` trait provides a polymorphic `summary` method. Each piece—generics, enums, traits—contributes a different kind of abstraction, and they compose cleanly.

18.6 The Future of Generics in Lattice

Lattice's generic system today is best described as *syntax-first*: the parser fully understands type parameters and type arguments, and the AST preserves them, but the compiler and runtime do not yet enforce them. This is a deliberate staged approach.

What works today:

- Generic type parameter syntax on functions, structs, and enums.
- Type arguments in type expressions (`Map<String, Int>`).
- Function type expressions (`Fn(Int) -> Bool`).
- Array type shorthand (`[Int]`).
- Phase-qualified type expressions.

What is planned for future versions:

- Compile-time type checking against declared parameters.
- Trait bounds on type parameters (`<T: Describable>`).
- Better error messages when types do not match at call sites.
- Tooling support: the language server already reads type annotations for hover information (see `src/lsp_server.c`).

Generics Do Not Prevent Type Errors (Yet)

Because type parameters are erased at runtime, passing a `String` where an `Int` is expected will not produce a compile-time error—you will get a runtime error when the value is used incorrectly. Use generics for documentation today, and look forward to enforcement in future Lattice versions.

Write your generic annotations now. They cost nothing at runtime, they make your code dramatically easier to understand, and when the type checker arrives, your code will already be ready.

18.7 Exercises

1. **Generic stack.** Define a `struct Stack<T>` with fields `items: [T]` and `size: Int`. Implement a `Pushable` trait with `push`, `pop`, and `peek` methods. Test it with both integers and strings.
2. **Option pipeline.** Write three functions: `map_option<T, U>`, `flat_map_option<T, U>` (where `f` returns an `Option`), and `or_else<T>` (provides a fallback option). Chain them to transform `Option::Some(5)` into `Option::Some("10")` (double, then convert to string).

3. **Generic key-value store.** Define `struct KVStore<K, V>` that wraps a `Map`. Implement `set`, `get`, and `has` methods. Use it to build a phone book mapping names to numbers.

4. **Type expression exploration.** Write a function whose signature uses all three type expression forms: a named type with arguments (`Map<String, Int>`), an array type (`[String]`), and a function type (`Fn(String) -> Int`). The function should accept a list of strings, a scoring function, and return a map from strings to their scores.

5. **Generic Result chain.** Define `enum Result<T, E>` and write a function `and_then<T, U, E>(r: Result, f: Fn(T) -> Result) -> Result` that applies `f` only if `r` is `Ok`. Chain three `and_then` calls to parse a string into an integer, double it, and validate that it is positive.

What's Next

With structs, enums, traits, and generics, you now have Lattice's full toolkit for structuring data and building abstractions. But programs do not just model data—they *do* things, often many things at once. In the next part we enter the world of *concurrency*: how Lattice's structured concurrency model with `scope` and `spawn` lets you run tasks in parallel while keeping your code safe and your sanity intact.

Part V

Concurrency

Chapter 19

Structured Concurrency with `scope` and `spawn`

Suppose you need to fetch three web pages at once, resize a batch of images in parallel, or crunch numbers across all your CPU cores. Most languages hand you a thread primitive and wish you luck—leaving you to track which threads are still running, which have crashed, and whether anyone remembered to join them. Lattice takes a different approach: every concurrent task is *scoped*, meaning it is born inside a block and *guaranteed* to finish before that block returns. No orphaned threads. No forgotten futures. No fire-and-forget surprises at 3 a.m.

Let's see what that looks like.

19.1 The `scope { spawn { . . . } }` Model

The two keywords you need are `scope` and `spawn`. A `scope` block creates a concurrency region; each `spawn` inside it launches a new OS thread:

Listing 19.1: Your first scope/spawn

```
scope {
  spawn {
    print("Hello from thread A")
  }
  spawn {
    print("Hello from thread B")
  }
  print("Hello from the parent")
}
print("Both spawns are finished by here")
```

Run this a few times and you will see the three messages interleaved in different orders—but the final `print` always executes *after* every spawn has completed. That is the fundamental contract: **a `scope` block does not return until every `spawn` inside it has finished**.

Structured Concurrency

Structured concurrency is a programming discipline in which concurrent tasks are confined to a lexical scope. The scope acts as a lifetime boundary: it waits for all child tasks to complete (or fail) before control flows past it. In Lattice, this is expressed with `scope { spawn { ... } }`.

19.1.1 What Goes Where

You can place any code inside a `scope` block, not only `spawn` statements. Non-spawn statements run *first*, on the parent thread, before any spawn threads are launched:

Listing 19.2: Synchronous setup inside a scope

```
scope {
  // This runs first, on the parent thread.
  let urls = ["https://example.com/a", "https://example.com/b"]
  print("Starting downloads for ${urls.len()} URLs")

  spawn {
    // Thread 1
    print("Downloading ${urls[0]}")
  }
  spawn {
    // Thread 2
    print("Downloading ${urls[1]}")
  }
}
```

The synchronous preamble is compiled into a separate body chunk and executed before the spawn threads are created. If the preamble fails, the spawns never start.

Scope Is an Expression

Like most constructs in Lattice, `scope` is an expression—but it always evaluates to `Unit`. If you need to collect results from spawned tasks, use a `Channel` or a `Ref`, both of which we will explore later in this chapter and in Chapter 20.

19.1.2 Multiple Spawns

There is no hard limit on the number of `spawn` blocks inside a single `scope`. Each one becomes its own OS thread:

Listing 19.3: Many spawns in one scope

```
scope {
  spawn { print("Worker 1") }
  spawn { print("Worker 2") }
  spawn { print("Worker 3") }
  spawn { print("Worker 4") }
}
// All four workers are done.
```

Under the hood, the compiler counts the `spawn` blocks at compile time and emits a single `OP_SCOPE` instruction that carries the spawn count as an inline operand. The VM then allocates a `VMSpawnTask` for each spawn, clones the virtual machine state, and calls `pthread_create` for each task—all visible in `src/stackvm.c`.

Keep Spawns Coarse-Grained

Each `spawn` creates a real OS thread with its own stack and cloned VM environment. Spawning thousands of micro-tasks is not the right pattern here—use iterators or channels for fine-grained parallelism. Think of `spawn` as "I have a meaningful chunk of work that should run on its own core."

19.2 Why Structured Concurrency Matters

If you have ever chased a bug involving a detached thread that outlived its parent function, you already know why structured concurrency matters. Let's make the argument concrete.

19.2.1 The Problem with Unstructured Threads

Consider pseudocode in a language with raw thread spawning:

Listing 19.4: An unstructured concurrency hazard (pseudocode)

```
// Danger: unstructured thread (NOT valid Lattice)
fn fetch_data() {
  let handle = thread_spawn(|| {
    // What if fetch_data returns before this finishes?
    // What if this panics and nobody catches it?
    let data = download("https://api.example.com/report")
    save_to_disk(data)
  })
  // Oops, we forgot to join the handle!
  return "done"
}
```

This pattern leads to three classes of bugs:

1. **Orphaned tasks.** The spawned thread continues after the function returns, possibly accessing freed memory or stale references.

2. **Silent failures.** If the thread panics, nobody is listening. Errors vanish into the void.

3. **Resource leaks.** File handles, network sockets, and memory held by the thread are never cleaned up if the parent forgets to join.

19.2.2 Lattice's Guarantee

Lattice eliminates all three by construction:

- You *cannot* spawn a thread outside a `scope`. The compiler rejects bare `spawn` blocks.
- Every `scope` waits for *all* spawns. There is no way to "forget" to join—it is automatic.
- If *any* spawn encounters a runtime error, the scope collects it and re-raises it in the parent. The first error wins; all threads are still joined before the error propagates.

Listing 19.5: Error propagation from spawns

```
try {
  scope {
    spawn {
      // This will fail
      let result = parse_int("not_a_number")
    }
    spawn {
      print("I still run to completion")
    }
  }
} catch err {
  print("Caught from spawn: ${err}")
}
```

The `try/catch` around the scope catches errors from *any* of the spawned threads. In the VM implementation (see `stackvm_spawn_thread_fn` in `src/stackvm.c`), each child VM stores its error string in the `VMSpawnTask.error` field; after all threads are joined, the parent collects the first non-null error and re-raises it.

Errors Don't Cancel Siblings

When one spawn encounters an error, the other spawns are *not* cancelled. They all run to completion (or to their own errors), and then the scope reports the first error. If you need cancellation, use a shared `Ref` as a signal flag that spawns check periodically—we will see this pattern in Section 19.4.

19.3 Joining Semantics—No Orphaned Tasks

Let's trace exactly what happens when a `scope` executes:

1. **Environment snapshot.** The VM pushes a new scope in the environment and exports all live local variables into it via `env_define`. Each spawned thread receives a *deep clone* of this environment—spawns see the values as they existed at the moment the scope began.

2. **Synchronous body.** Any non-spawn statements run on the parent thread.

3. **Thread creation.** For each spawn, the VM calls `stackvm_clone_for_thread` to create an independent child VM, then `pthread_create` to launch the thread. Each child VM gets its own stack, heap (`DualHeap`), and garbage collector.

4. **Join barrier.** The parent calls `pthread_join` on every spawn thread in order. This is an unconditional, blocking join—the parent cannot proceed until all children are done.

5. **Cleanup.** Child VMs are freed, the environment scope is popped, and errors (if any) are propagated.

Listing 19.6: Demonstrating join semantics with timing

```
fn slow_work(label: String, ms: Int) {
  // Simulate work
  let start = clock()
  while clock() - start < ms {
    // busy-wait
  }
  print("${label} finished after ~${ms}ms")
}

scope {
  spawn { slow_work("Task A", 200) }
  spawn { slow_work("Task B", 100) }
  spawn { slow_work("Task C", 300) }
}
print("All tasks done")
// Output (order of first three lines varies):
// Task B finished after ~100ms
// Task A finished after ~200ms
// Task C finished after ~300ms
// All tasks done
```

The key observation: `"All tasks done"` always prints last, regardless of how fast each spawn completes.

19.3.1 Nested Scopes

Scopes can nest. Each inner scope is a complete join barrier:

Listing 19.7: Nested scopes

```
scope {
  spawn {
    print("Outer spawn starts")
    scope {
      spawn { print("Inner spawn A") }
      spawn { print("Inner spawn B") }
    }
    // Inner spawns are done here.
    print("Outer spawn continues")
  }
}
print("Everything is done")
```

The inner `scope` ensures its two spawns finish before the outer spawn continues. The outer `scope` then ensures the outer spawn finishes before the final `print`. This nesting composes cleanly—you never have to reason about which thread outlives which.

19.3.2 Scopes in Loops

A common pattern is placing a `scope` inside a loop to process batches in parallel:

Listing 19.8: Scope inside a loop

```
let batches = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
]

for batch in batches {
  scope {
    spawn {
      let sum = batch.reduce(0, |acc, x| acc + x)
      print("Batch sum: ${sum}")
    }
  }
}
// Each batch is fully processed before the next one starts.
```

Each iteration's `scope` is a self-contained concurrency region. This is safe because the scope joins before the loop advances to the next iteration.

19.4 Sharing Data Between Spawned Tasks

Here is the rule you need to internalize: **each spawn gets a deep clone of the parent's variables**. Mutations inside a spawn do *not* affect the parent or sibling spawns.

Listing 19.9: Spawns get independent copies

```
flux counter = 0
scope {
  spawn {
    counter = counter + 1
    print("Spawn A sees: ${counter}")  // 1
  }
  spawn {
    counter = counter + 1
    print("Spawn B sees: ${counter}")  // 1
  }
}
print("Parent sees: ${counter}")  // 0
```

Both spawns start with `counter = 0` (cloned from the parent), increment their own copy, and print `1`. The parent's `counter` remains `0`.

Why Deep Cloning?

Deep cloning prevents data races by giving each thread its own copy of the world. In the implementation (`src/stackvm.c`, `stackvm_export_locals_to_env`), the parent's live locals are cloned via `value_deep_clone` into the child VM's environment. This means spawns can freely mutate their local state without synchronization.

19.4.1 Communicating Through Channels

If you *do* need spawns to communicate, use a `Channel`. Channels are thread-safe by design—they use a mutex-protected buffer internally (see `src/channel.c`).

Listing 19.10: Spawns communicating via a channel

```
let results = Channel::new()

scope {
  spawn {
    let answer = 21 * 2
    results.send(answer)
  }
  spawn {
    let answer = 6 * 7
    results.send(answer)
  }
}

// Both spawns are done; collect results.
print(results.recv())  // 42
print(results.recv())  // 42
```

Because `Channel` values are reference-counted and mutex-protected, they can be safely shared across spawn boundaries. When the parent's environment is cloned for each spawn, the channel itself is *not* deep-copied—both spawns and the parent hold references to the same underlying `LatChannel`, with the reference count incremented via `channel_retain`.

We will explore channels in depth in Chapter 20.

19.4.2 Communicating Through Ref

For shared mutable state that doesn't need channel semantics, Lattice offers `Ref`—a reference-counted, thread-safe wrapper:

Listing 19.11: Using Ref for shared state

```
let total = Ref::new(0)

scope {
  spawn {
    let current = total.get()
    total.set(current + 10)
  }
  spawn {
    let current = total.get()
    total.set(current + 20)
  }
}

print("Total: ${total.get()}")
// Could be 10, 20, or 30 depending on timing
```

Ref Is Not Atomic

A `Ref` provides shared mutable state, but it does *not* provide atomicity. The `get()` followed by `set()` pattern above is a classic race condition. If you need atomic read-modify-write semantics, use a channel to serialize access, or restructure your code so that each spawn writes to its own channel and the parent aggregates. We will cover safe patterns in Chapter 21.

19.5 The Sublimate Phase for Thread-Safe Values

In Chapter 11, we met the **fluid** and **crystal** phases—values that are mutable or frozen, respectively. Concurrency introduces a third phase transition: **sublimated**.

The Sublimated Phase

A *sublimated* value is one that has been marked as thread-safe and immutable. Like a crystal value, a sublimated value cannot be mutated. Unlike crystal, sublimation is specifically intended for data that will cross thread boundaries. The name comes from chemistry: sublimation is the phase transition from solid directly to gas—the value "evaporates" from one thread's local world into the shared concurrent space.

19.5.1 Using `sublimate`

The `sublimate` keyword transitions a value to the sublimated phase:

Listing 19.12: Sublimating a value

```
flux config = [1, 2, 3]

// Mark as thread-safe before sharing
sublimate config

scope {
  spawn {
    // config is sublimated --- safe to read, cannot mutate
    print("Config length: ${config.len()}")
  }
  spawn {
    print("First element: ${config[0]}")
  }
}
```

Once sublimated, any attempt to mutate the value raises a runtime error:

Listing 19.13: Sublimated values cannot be mutated

```
flux data = [10, 20, 30]
sublimate data
// data.push(40)  // Runtime error: cannot modify a sublimated value
```

19.5.2 Sublimate vs. Freeze

You might wonder: "If sublimated values are immutable, how are they different from frozen values?"

- `freeze` produces a **crystal** value. Crystals are immutable but are designed for single-threaded lifetime management—they participate in arenas and crystal regions.

- `sublimate` produces a **sublimated** value. Sublimated values are immutable *and* conceptually detached from any single thread's memory region. They are the right choice for data that will be read by multiple threads.

In the VM implementation (`src/stackvm.c`), both crystal and sublimated values are treated identically for mutation checks—any attempt to set a field, push to an array, or assign an index is rejected. The `OP_SUBLIMATE_VAR` opcode sets the value's phase tag to `VTAG_SUBLIMATED` and fires any reactive bonds.

Listing 19.14: Sublimate and freeze comparison

```
flux temperatures = [72.1, 68.4, 75.0]

// Option 1: freeze --- good for single-threaded immutability
fix frozen_temps = freeze temperatures

// Option 2: sublimate --- good for cross-thread sharing
flux shared_temps = [72.1, 68.4, 75.0]
sublimate shared_temps

scope {
  spawn {
    // Both are readable, neither is mutable
    print("Frozen: ${frozen_temps[0]}")
    print("Sublimated: ${shared_temps[0]}")
  }
}
```

When to Sublimate

Use `sublimate` when you have configuration data, lookup tables, or shared constants that multiple spawns need to read. It communicates intent: "this value is intended for concurrent access." If you are not sharing data across threads, plain `freeze` is sufficient and more conventional.

19.5.3 Sublimating Complex Structures

Sublimation is deep—it applies to the entire value graph:

Listing 19.15: Deep sublimation

```
flux settings = Map::new()
settings["db_host"] = "localhost"
settings["db_port"] = 5432
settings["retries"] = 3

sublimate settings

scope {
  spawn {
    print("Connecting to ${settings["db_host"]}:${settings["db_port"]}")
  }
  spawn {
    print("Max retries: ${settings["retries"]}")
  }
}
```

After sublimation, neither the map nor any of its values can be modified—the entire structure is safe for concurrent reads.

19.6 Practical Examples

Let's put everything together with some real-world-inspired examples.

19.6.1 Parallel Computation: Fan-Out / Fan-In

The classic concurrent pattern: split work across threads, then collect results.

Listing 19.16: Fan-out/fan-in with channels

```
fn compute_chunk(numbers: Array, result_ch: Channel) {
  let sum = numbers.reduce(0, |acc, n| acc + n)
  result_ch.send(sum)
}

let data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
let results = Channel::new()

// Split into 3 chunks and process in parallel
scope {
  spawn { compute_chunk(data.slice(0, 4), results) }
  spawn { compute_chunk(data.slice(4, 8), results) }
  spawn { compute_chunk(data.slice(8, 12), results) }
}

// Collect and sum the partial results
let total = results.recv() + results.recv() + results.recv()
print("Total: ${total}")  // Total: 78
```

This is a textbook fan-out/fan-in. The `scope` ensures all three workers complete before we start receiving. The channel collects partial sums, and we aggregate them on the parent thread.

19.6.2 Parallel String Processing

Listing 19.17: Processing files in parallel

```
fn process_name(name: String, output: Channel) {
  let processed = name.trim().upper()
  output.send(processed)
}

let names = ["  alice  ", "BOB", " Charlie ", "  dave"]
let output = Channel::new()

scope {
  spawn { process_name(names[0], output) }
  spawn { process_name(names[1], output) }
  spawn { process_name(names[2], output) }
  spawn { process_name(names[3], output) }
}

// Collect all results
flux processed_names = []
for _ in 0..4 {
  processed_names.push(output.recv())
}
print(processed_names)
// ["ALICE", "BOB", "CHARLIE", "DAVE"] (order may vary)
```

19.6.3 Nested Parallelism: Two-Level Fan-Out

Listing 19.18: Nested scope for two-level parallelism

```
fn process_region(region: String, cities: Array, results: Channel) {
  let city_results = Channel::new()

  scope {
    spawn {
      city_results.send("${cities[0]}: processed")
    }
    spawn {
      city_results.send("${cities[1]}: processed")
    }
  }

  let summary = "${region}: ${city_results.recv()}, ${city_results.recv()}"
  results.send(summary)
}

let results = Channel::new()

scope {
  spawn {
    process_region("West", ["Seattle", "Portland"], results)
  }
  spawn {
    process_region("East", ["Boston", "New York"], results)
  }
}

print(results.recv())
print(results.recv())
```

Each region processes its cities in parallel (inner `scope`), and regions themselves run in parallel (outer `scope`). The structured concurrency model guarantees that inner scopes complete before their parent spawn returns, which completes before the outer scope returns.

19.6.4 Using a Cancellation Flag

Since spawns cannot be forcibly cancelled, we use a shared `Ref` as a cooperative cancellation signal:

Listing 19.19: Cooperative cancellation with Ref

```
let should_stop = Ref::new(false)
let results = Channel::new()

scope {
  spawn {
    // Producer: generate values until told to stop
    flux count = 0
    while should_stop.get() == false {
      count = count + 1
      results.send(count)
      if count >= 100 {
        break
      }
    }
    results.close()
  }
  spawn {
    // Consumer: read values, signal stop after finding 42
    flux done = false
    while done == false {
      let val = results.recv()
      if val == 42 {
        print("Found 42! Signaling stop.")
        should_stop.set(true)
        done = true
      }
    }
  }
}
```

The `Ref` serves as a boolean flag visible to both threads. The consumer sets it when it has seen enough, and the producer checks it each iteration. This is a cooperative, structured alternative to thread cancellation.

19.6.5 Scope with Error Handling

Listing 19.20: Handling errors from parallel tasks

```
fn safe_divide(a: Int, b: Int, results: Channel) {
  if b == 0 {
    results.send("error: division by zero")
  } else {
    results.send(to_string(a / b))
  }
}

let results = Channel::new()

scope {
  spawn { safe_divide(100, 5, results) }
  spawn { safe_divide(200, 0, results) }
  spawn { safe_divide(300, 3, results) }
}

for _ in 0..3 {
  print(results.recv())
}
// Output (order may vary):
// 20
// error: division by zero
// 100
```

When errors are expected and recoverable, handling them inside the spawn and communicating results through a channel is often cleaner than relying on `try`/`catch` around the scope.

19.7 Under the Hood

For readers who want to understand the machinery, here is a tour of how `scope` and `spawn` are compiled and executed.

19.7.1 Compilation

The compiler (`src/stackcompiler.c`) handles `scope` expressions in `EXPR_SCOPE`:

1. It counts the `spawn` blocks and separates them from the synchronous statements.

2. The synchronous statements are compiled into a *sub-body chunk*—a self-contained bytecode chunk that can be executed independently.

3. Each `spawn` body is also compiled into its own sub-body chunk.

4. A single `OP_SCOPE` instruction is emitted with the spawn count, the synchronous body index, and each spawn body index as inline operands.

If there are no `spawn` blocks (a bare `scope { ... }`), the compiler emits `OP_SCOPE` with a spawn count of zero, and the body runs synchronously as a sub-chunk—useful for isolating variable scopes.

19.7.2 Execution

When the VM encounters `OP_SCOPE` (`src/stackvm.c`):

1. It pushes a new environment scope and exports all live locals via deep cloning.

2. If `spawn_count == 0`, it runs the body chunk synchronously and returns the result.

3. Otherwise, it runs the synchronous body first, then allocates a `VMSpawnTask` array.

4. For each spawn, `stackvm_clone_for_thread` creates an independent child `StackVM` with its own runtime, heap, and GC state. The function `stackvm_export_locals_to_env` copies the parent's locals into the child's environment.

5. Each child is launched with `pthread_create`. The thread function `stackvm_spawn_thread_fn` sets up a thread-local `DualHeap`, runs the spawn chunk, and records any error.

6. The parent calls `pthread_join` on every thread, collects errors, frees child VMs, and pops the environment scope.

Thread-Local Heaps

Each spawn thread gets its own `DualHeap` for memory allocation. This means spawned tasks can allocate freely without contending on a global allocator lock. The heap is freed when the thread completes.

19.8 Exercises

1. **Parallel Sum.** Given an array of 1000 integers, write a program that splits the array into 4 chunks, sums each chunk in a separate spawn, and then adds the partial sums together. Use a `Channel` to collect results.

2. **Scope Ordering.** Write a program with a scope containing three spawns that each print a message. Run it several times and observe the output order. Then add a fourth print statement *after* the scope. Verify that it always prints last.

3. **Nested Scope Hierarchy.** Create a three-level nested scope structure: an outer scope with two spawns, each of which contains an inner scope with two more spawns. Each innermost spawn should print its "address" (e.g., `"Outer 1, Inner 2"`). Verify that all messages print before the program exits.

4. **Cancellation Pattern.** Implement a producer-consumer pair where the producer generates random strings and sends them through a channel. The consumer reads strings and stops the producer (using a `Ref` flag) when it finds one longer than 10 characters.

5. **Error Propagation.** Write a scope with three spawns where the second one deliberately causes an error. Wrap the scope in `try`/`catch` and verify that the error is caught. Confirm that the other spawns still ran to completion by having them write to a channel.

What's Next

We have seen how `scope` and `spawn` give you safe, structured concurrency with automatic joining and error propagation. But we have only scratched the surface of the `Channel`—we used it as a convenient mailbox without exploring its full API. In the next chapter, we will dive deep into channels: how to create them, how `send` and `recv` work, how to multiplex across multiple channels with `select`, and how to build powerful data pipelines that transform values as they flow between threads.

Chapter 20

Channels and `select`

In the previous chapter, we used channels as convenient mailboxes—toss a value in one end, pull it out the other. But channels in Lattice are a full concurrency primitive: they are the primary mechanism for safe communication between threads, and with `select`, they become a powerful multiplexing tool that lets you wait on multiple sources of data simultaneously.

Let's start from the beginning.

20.1 `Channel::new()`, `send()`, `recv()`, `close()`

A channel is a thread-safe FIFO queue. You create one with `Channel::new()`, send values into it, and receive values from it:

Listing 20.1: Channel basics

```
let ch = Channel::new()

ch.send(42)
ch.send("hello")
ch.send(true)

print(ch.recv())  // 42
print(ch.recv())  // hello
print(ch.recv())  // true
```

Channels are untyped—you can send any Lattice value through them. Values come out in the order they were sent (first-in, first-out).

Channel

A *Channel* is an unbounded, thread-safe FIFO queue. It supports three core operations: `send(value)` enqueues a value, `recv()` dequeues a value (blocking if the channel is empty), and `close()` signals that no more values will be sent. Internally, a channel is a mutex-protected vector with a condition variable for blocking receives (see `include/channel.h`).

20.1.1 `send()` — Putting Values In

The `send()` method enqueues a value into the channel's buffer:

Listing 20.2: Sending values

```
let orders = Channel::new()

orders.send("latte")
orders.send("cappuccino")
orders.send("espresso")

print("Queued ${3} orders")
```

Sending is non-blocking: because Lattice channels are unbounded, `send()` always succeeds immediately (as long as the channel is open). Under the hood, `channel_send` in `src/channel.c` acquires the channel's mutex, pushes the value onto the internal `LatVec` buffer, signals the `cond_notempty` condition variable to wake any blocked receivers, and also wakes any select waiters.

Sending to a Closed Channel

If you `send()` to a closed channel, the value is silently dropped and the send returns `false` internally. The value is freed to prevent memory leaks. Design your programs so that producers stop sending before or shortly after the channel is closed.

20.1.2 `recv()` — Taking Values Out

The `recv()` method dequeues the next value from the channel:

Listing 20.3: Receiving values

```
let ch = Channel::new()
ch.send(100)
ch.send(200)

let first = ch.recv()   // 100
let second = ch.recv()  // 200
print("Got ${first} and ${second}")
```

Here is the critical behavior: **if the channel is empty and still open, `recv()` blocks the calling thread until a value is available**. This blocking behavior is what makes channels useful for synchronization—a consumer naturally waits for its producer.

Listing 20.4: Blocking recv

```
let ch = Channel::new()

scope {
  spawn {
    // Simulate slow work
    let start = clock()
    while clock() - start < 100 { }
    ch.send("result ready")
  }
  spawn {
    // This recv blocks until the value arrives
    let result = ch.recv()
    print("Got: ${result}")
  }
}
```

In the implementation (`src/channel.c`, `channel_recv`), receiving works by locking the mutex and entering a `while` loop that calls `pthread_cond_wait` as long as the buffer is empty and the channel is not closed. When a value is available, it is shifted from the front of the buffer (FIFO semantics) and returned.

20.1.3 What Happens When You `recv()` from a Closed, Empty Channel?

If the channel is closed and its buffer is empty, `recv()` returns `nil`. This provides a clean way to detect that a producer is done:

Listing 20.5: Receiving from a closed channel

```
let ch = Channel::new()
ch.send(1)
ch.send(2)
ch.close()

print(ch.recv())  // 1
print(ch.recv())  // 2
print(ch.recv())  // nil (channel closed and empty)
```

20.1.4 `close()` — Signaling Completion

The `close()` method signals that no more values will be sent:

Listing 20.6: Closing a channel

```
let tasks = Channel::new()

tasks.send("task_a")
tasks.send("task_b")
tasks.close()

// Receivers can still drain buffered values
print(tasks.recv())  // task_a
print(tasks.recv())  // task_b
print(tasks.recv())  // nil
```

Closing is a one-way operation—once closed, a channel stays closed. The implementation (`channel_close` in `src/channel.c`) sets the `closed` flag, then broadcasts on the condition variable to wake all blocked receivers and select waiters. This ensures no thread stays blocked forever on a closed channel.

Always Close Producer Channels

When a producer is done sending, close the channel. This lets consumers detect the end of the stream and exit their loops cleanly. A channel that is never closed will cause any `recv()` call to block indefinitely once the buffer is drained.

20.1.5 Channel Lifecycle

Channels are reference-counted. When you share a channel across spawns, each copy increments the reference count (`channel_retain`). When a copy goes out of scope, the count is decremented (`channel_release`). The channel's internal buffer is freed only when the last reference is released.

Listing 20.7: Channel lifecycle across spawns

```
fn producer(ch: Channel, items: Array) {
  for item in items {
    ch.send(item)
  }
  ch.close()
}

fn consumer(ch: Channel) {
  flux item = ch.recv()
  while item != nil {
    print("Consumed: ${item}")
    item = ch.recv()
  }
}

let pipeline = Channel::new()

scope {
  spawn { producer(pipeline, ["apple", "banana", "cherry"]) }
  spawn { consumer(pipeline) }
}
// Output:
// Consumed: apple
// Consumed: banana
// Consumed: cherry
```

20.2 Producer-Consumer Patterns

The producer-consumer pattern is the bread and butter of channel-based concurrency. One thread produces data, another consumes it, and the channel decouples them.

20.2.1 Single Producer, Single Consumer

The most common variant:

Listing 20.8: Single producer, single consumer

```
let work_queue = Channel::new()

scope {
  // Producer
  spawn {
    for i in 0..10 {
      work_queue.send(i * i)
    }
    work_queue.close()
  }

  // Consumer
  spawn {
    flux val = work_queue.recv()
    while val != nil {
      print("Processing: ${val}")
      val = work_queue.recv()
    }
    print("Consumer done")
  }
}
```

The consumer loop is idiomatic: call `recv()`, check for `nil` (which signals the channel is closed and drained), and process each value.

20.2.2 Multiple Producers, Single Consumer

Multiple spawns can send to the same channel:

Listing 20.9: Multiple producers, single consumer

```
let log_channel = Channel::new()

scope {
  spawn {
    for i in 0..5 {
      log_channel.send("[Server A] Request ${i}")
    }
  }
  spawn {
    for i in 0..5 {
      log_channel.send("[Server B] Request ${i}")
    }
  }
  spawn {
    // Give producers a moment, then close
    let start = clock()
    while clock() - start < 50 { }
    log_channel.close()
  }
}

// Drain the log
flux msg = log_channel.recv()
while msg != nil {
  print(msg)
  msg = log_channel.recv()
}
```

Because the channel is mutex-protected, sends from different threads are serialized automatically—no data corruption, no lost messages.

20.2.3 Single Producer, Multiple Consumers

This pattern distributes work across consumers:

Listing 20.10: Work distribution across consumers

```
fn worker(id: Int, jobs: Channel, results: Channel) {
  flux job = jobs.recv()
  while job != nil {
    let result = "Worker ${id} processed job ${job}"
    results.send(result)
    job = jobs.recv()
  }
}

let jobs = Channel::new()
let results = Channel::new()

// Enqueue work
for i in 0..12 {
  jobs.send(i)
}
jobs.close()

scope {
  spawn { worker(1, jobs, results) }
  spawn { worker(2, jobs, results) }
  spawn { worker(3, jobs, results) }
}

// Collect all results
for _ in 0..12 {
  print(results.recv())
}
```

Each worker calls `recv()` on the same jobs channel. Because `recv()` is atomic (only one thread gets each value), the work is naturally distributed—no explicit load balancing needed.

20.3 `select` with Multiple Channels

What if you need to wait on *multiple* channels at once? Maybe you have two data sources and want to process whichever one has data ready first. This is where `select` comes in.

Listing 20.11: Basic select

```
let weather = Channel::new()
let news = Channel::new()

scope {
  spawn { weather.send("Sunny, 72F") }
  spawn { news.send("Lattice 1.0 released!") }
}

let result = select {
  weather -> report { "Weather: ${report}" }
  news -> headline { "News: ${headline}" }
}
print(result)
// Either "Weather: Sunny, 72F" or "News: Lattice 1.0 released!"
```

A `select` expression waits on multiple channels and executes the arm corresponding to the first channel that has data available.

`select` Expression

A `select` expression monitors multiple channels simultaneously and executes the body of the first arm that successfully receives a value. If multiple channels have data ready, one is chosen at random (via Fisher-Yates shuffle in the VM) to ensure fairness. The expression evaluates to the result of the chosen arm's body.

20.3.1 Anatomy of a `select` Arm

Each arm of a `select` has three parts:

Listing 20.12: Select arm anatomy

```
select {
  channel_expr -> binding_name { body }
}
```

- `channel_expr`: An expression that evaluates to a `Channel`.
- `binding_name`: A variable name that will hold the received value inside the body. This is optional—you can omit the binding if you don't need the value.

- body: The code to execute when this channel delivers a value.

20.3.2 Fairness

When multiple channels have data ready simultaneously, Lattice does not always pick the first arm in source order. Instead, the VM shuffles the arm indices using the Fisher-Yates algorithm before polling. This prevents starvation: a fast producer on one channel cannot permanently starve a slower producer on another.

Listing 20.13: Fairness in select

```
let fast = Channel::new()
let slow = Channel::new()

// Load both channels
for i in 0..10 {
  fast.send("fast-${i}")
  slow.send("slow-${i}")
}

// Each select picks fairly from both
for _ in 0..5 {
  let result = select {
    fast -> msg { msg }
    slow -> msg { msg }
  }
  print(result)
}
// You'll see a mix of "fast-*" and "slow-*" messages
```

20.3.3 Select in a Loop

A common pattern is to use `select` inside a loop to continuously process events from multiple sources:

Listing 20.14: Event loop with select

```
let keyboard = Channel::new()
let network = Channel::new()
let timer = Channel::new()

// Simulate event sources
scope {
  spawn {
    keyboard.send("key_press: Enter")
    keyboard.send("key_press: Escape")
    keyboard.close()
  }
  spawn {
    network.send("packet: 200 OK")
    network.close()
  }
  spawn {
    timer.send("tick")
    timer.send("tick")
    timer.close()
  }
}

// Process all events
flux running = true
while running {
  let event = select {
    keyboard -> key { "Keyboard: ${key}" }
    network -> pkt { "Network: ${pkt}" }
    timer -> t { "Timer: ${t}" }
    default { running = false; "done" }
  }
  print(event)
}
```

The `default` arm fires when all channels are empty (or closed), allowing us to break out of the loop cleanly.

20.4 `default` and `timeout` Arms

Plain `select` blocks until one of its channel arms can receive. Two special arms modify this behavior.

20.4.1 The `default` Arm

The `default` arm executes immediately if no channel has data ready:

Listing 20.15: Select with default

```
let ch = Channel::new()

let result = select {
  ch -> val { "Got: ${val}" }
  default { "Nothing available" }
}
print(result)  // Nothing available
```

This turns `select` into a non-blocking check. The `default` arm is useful for polling patterns where you want to do other work when no messages are ready.

Listing 20.16: Polling pattern with default

```
let inbox = Channel::new()

scope {
  spawn {
    let start = clock()
    while clock() - start < 100 { }
    inbox.send("delayed message")
  }
  spawn {
    flux attempts = 0
    flux found = false
    while found == false {
      let result = select {
        inbox -> msg {
          found = true
          msg
        }
        default {
          attempts = attempts + 1
          "waiting..."
        }
      }
      if found {
        print("Got message after ${attempts} polls: ${result}")
      }
    }
  }
}
```

Busy-Waiting with Default

A `select` with a `default` arm never blocks. If you put it in a tight loop, you will burn CPU cycles polling. Consider using a `timeout` arm instead if you can tolerate some latency.

20.4.2 The `timeout` Arm

The `timeout` arm takes a duration in milliseconds and fires if no channel delivers a value within that time:

Listing 20.17: Select with timeout

```
let slow_service = Channel::new()

let result = select {
  slow_service -> data { "Got: ${data}" }
  timeout 500 { "Timed out after 500ms" }
}
print(result)  // Timed out after 500ms
```

The timeout is implemented using `pthread_cond_timedwait` in the VM (see `src/stackvm.c`, the `OP_SELECT` handler). The VM computes an absolute deadline from the current time plus the timeout value, then waits on the shared condition variable with that deadline.

Listing 20.18: Timeout with fallback logic

```
fn fetch_with_timeout(source: Channel, timeout_ms: Int) {
  let result = select {
    source -> data { data }
    timeout timeout_ms { nil }
  }
  if result == nil {
    print("Request timed out, using cached data")
    return "cached_value"
  }
  return result
}

let api = Channel::new()

// Simulate a slow API
scope {
  spawn {
    let start = clock()
    while clock() - start < 1000 { }
    api.send("fresh data")
  }
  spawn {
    let value = fetch_with_timeout(api, 200)
    print("Using: ${value}")
  }
}
// Output: Request timed out, using cached data
//         Using: cached_value
```

20.4.3 Combining `default` and `timeout`

You can have *either* a `default` or a `timeout` arm, but not both. If a `default` arm is present, it always takes priority over any timeout since it fires immediately when no data is ready.

Choosing Between `default` and `timeout`

- Use `default` when you want to do other work *immediately* if no message is ready (non-blocking poll).
- Use `timeout` when you can afford to wait a bit but need an upper bound on how long (bounded blocking).
- Use neither when you are happy to wait indefinitely for a message (unbounded blocking).

20.4.4 Select with All Channels Closed

If all channel arms refer to closed, empty channels:

- If a `default` arm is present, it executes.
- If a `timeout` arm is present, the select waits until the timeout, then executes the timeout arm.
- If neither is present, the select unblocks and returns `Unit`.

Listing 20.19: Select with closed channels

```
let ch = Channel::new()
ch.close()

let result = select {
  ch -> val { "Got: ${val}" }
  default { "All channels closed" }
}
print(result)  // All channels closed
```

20.5 Building Pipelines with Channels

One of the most elegant uses of channels is building data processing pipelines, where each stage reads from one channel, transforms the data, and writes to another.

20.5.1 A Two-Stage Pipeline

Listing 20.20: Two-stage channel pipeline

```
fn generate(output: Channel) {
  for i in 1..11 {
    output.send(i)
  }
  output.close()
}

fn square(input: Channel, output: Channel) {
  flux val = input.recv()
  while val != nil {
    output.send(val * val)
    val = input.recv()
  }
  output.close()
}

let stage1 = Channel::new()
let stage2 = Channel::new()

scope {
  spawn { generate(stage1) }
  spawn { square(stage1, stage2) }
}

// Consume the final output
flux result = stage2.recv()
while result != nil {
  print(result)
  result = stage2.recv()
}
// Output: 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
```

Each stage runs concurrently. As soon as the generator produces a value, the squarer can consume and transform it—the pipeline processes data in a streaming fashion rather than waiting for all data to be generated first.

20.5.2 A Three-Stage Pipeline

Let's add a filtering stage:

Listing 20.21: Three-stage pipeline: generate, filter, format

```
fn generate_numbers(output: Channel, limit: Int) {
  for i in 1..limit {
    output.send(i)
  }
  output.close()
}

fn filter_evens(input: Channel, output: Channel) {
  flux val = input.recv()
  while val != nil {
    if val % 2 == 0 {
      output.send(val)
    }
    val = input.recv()
  }
  output.close()
}

fn format_output(input: Channel, output: Channel) {
  flux val = input.recv()
  while val != nil {
    output.send("Even number: ${val}")
    val = input.recv()
  }
  output.close()
}

let raw = Channel::new()
let filtered = Channel::new()
let formatted = Channel::new()

scope {
  spawn { generate_numbers(raw, 21) }
  spawn { filter_evens(raw, filtered) }
  spawn { format_output(filtered, formatted) }
}

flux msg = formatted.recv()
while msg != nil {
  print(msg)
  msg = formatted.recv()
}
// Output:
// Even number: 2
// Even number: 4
// ...
// Even number: 20
```

20.5.3 Fan-Out Pipeline

You can split a pipeline by having multiple consumers read from the same channel:

Listing 20.22: Fan-out: two workers consuming one channel

```
fn heavy_compute(id: Int, input: Channel, output: Channel) {
  flux val = input.recv()
  while val != nil {
    // Simulate computation
    output.send("Worker ${id}: ${val * val}")
    val = input.recv()
  }
}

let input = Channel::new()
let output = Channel::new()

for i in 1..21 {
  input.send(i)
}
input.close()

scope {
  spawn { heavy_compute(1, input, output) }
  spawn { heavy_compute(2, input, output) }
}

for _ in 0..20 {
  print(output.recv())
}
```

Work is distributed across the two workers automatically because each `recv()` is atomic—only one worker gets each value.

20.6 Async Iterators: `async_iter`, `async_map`, `async_filter`

Building pipelines with raw channels is powerful but verbose. Lattice provides three built-in functions that bridge channels and iterators, giving you a more ergonomic way to build concurrent data flows.

20.6.1 async_iter — Channel-Backed Iterators

async_iter takes a closure that receives a channel, spawns the closure on a background thread, and returns an iterator that yields values from that channel:

Listing 20.23: Creating an async iterator

```
let numbers = async_iter(|ch| {
  for i in 1..6 {
    ch.send(i * 10)
  }
  // Channel is automatically closed when the closure returns
})

for val in numbers {
  print(val)
}
// Output:
// 10
// 20
// 30
// 40
// 50
```

async_iter

async_iter(producer_fn) creates a channel, spawns a detached thread running producer_fn with the channel as its argument, and returns an iterator that pulls values from the channel. When the producer closure returns, the channel is automatically closed, which causes the iterator to terminate.

Under the hood (src/runtime.c, native_async_iter), the function creates a LatChannel, clones the current VM for thread safety, and launches a detached thread that calls the closure with the channel. The returned iterator wraps the channel and calls channel_recv on each next().

20.6.2 async_map — Transform an Async Stream

async_map applies a transformation function to each element of an iterator:

Listing 20.24: Async map transformation

```
let raw_data = async_iter(|ch| {
  for i in 1..6 {
    ch.send(i)
  }
})

let doubled = async_map(raw_data, |x| x * 2)

for val in doubled {
  print(val)
}
// Output: 2, 4, 6, 8, 10
```

The transformation runs synchronously on the consumer thread as values are pulled—this is *not* an additional concurrent stage. If you need the transformation itself to run concurrently, use an explicit channel pipeline as shown in Section 20.5.

20.6.3 `async_filter` — Filter an Async Stream

`async_filter` keeps only elements that pass a predicate:

Listing 20.25: Async filter

```
let source = async_iter(|ch| {
  for i in 1..21 {
    ch.send(i)
  }
})

let evens = async_filter(source, |x| x % 2 == 0)

for val in evens {
  print(val)
}
// Output: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

20.6.4 Composing Async Operations

These functions compose naturally into a pipeline:

Listing 20.26: Composed async pipeline

```
let pipeline = async_iter(|ch| {
  for i in 1..101 {
    ch.send(i)
  }
})

let result = async_filter(
  async_map(pipeline, |x| x * x),
  |x| x % 3 == 0
)

flux count = 0
for val in result {
  count = count + 1
}
print("Found ${count} perfect squares divisible by 3")
```

This creates a three-stage processing flow: generate numbers 1–100, square them, then keep only those divisible by 3. The generation runs on a background thread; the mapping and filtering run on the consumer thread as values are pulled through.

When to Use Async Iterators vs. Raw Channels

Use `async_iter` when you have a single producer and want to consume its output with familiar `for` loop syntax. Use raw channels when you need multiple producers, multiple consumers, bidirectional communication, or explicit `select` multiplexing.

20.7 Under the Hood: `select` Implementation

For those curious about the machinery, here is how `select` works inside the VM.

20.7.1 Compilation

The compiler (`src/stackcompiler.c`, `EXPR_SELECT`) emits a single `OP_SELECT` instruction followed by per-arm metadata:

1. **Arm count** (1 byte): how many arms the select has.
2. For each arm, four bytes:

- **Flags**: bit 0 = default, bit 1 = timeout, bit 2 = has binding name.
- **Channel/timeout index**: index of a sub-chunk constant that evaluates to the channel (or timeout value).
- **Body index**: index of the sub-chunk for the arm's body.
- **Binding index**: index of the string constant for the binding name (or `0xFF` if no binding).

20.7.2 Execution

The VM's `OP_SELECT` handler (`src/stackvm.c`) follows this algorithm:

1. Export locals to a new environment scope (so sub-chunks can see them).
2. Evaluate all channel expressions to obtain `LatChannel*` pointers.
3. Evaluate the timeout expression (if present) to get a millisecond value.
4. Build a shuffled index array (Fisher-Yates) for fairness.
5. Enter a loop:
   (a) Try a non-blocking `channel_try_recv` on each channel arm (in shuffled order).
   (b) If a value is received: bind it in the environment, execute the arm body, and break.
   (c) If all channels are closed: execute the default arm (if present) and break.
   (d) If a default arm exists: execute it immediately (non-blocking) and break.
   (e) Otherwise: register a `LatSelectWaiter` on all channels, then `pthread_cond_wait` (or `pthread_cond_timedwait` if there is a timeout).
   (f) On wake: remove waiters and retry from step (a).
6. Clean up: release channels, pop scope, push result.

The `LatSelectWaiter` structure (defined in `include/channel.h`) is a linked-list node containing a shared mutex and condition variable. When any monitored channel receives data or is closed, it signals the waiter's condition variable, waking the select loop.

20.8 Exercises

1. **Ping-Pong.** Create two channels and two spawns that play ping-pong: spawn A sends "ping" on channel 1 and waits for "pong" on channel 2; spawn B does the reverse. Run 10 rounds and print each exchange.

2. **Timeout Handler.** Write a function that takes a channel and a timeout in milliseconds. Using `select`, return the first value from the channel or the string `"timeout"` if the deadline passes. Test with both fast and slow producers.

3. **Channel Merge.** Write a function `merge` that takes two channels and returns a new channel containing all values from both, interleaved in arrival order. Use `select` in a loop with a `default` arm to detect when both channels are drained.

4. **Async Pipeline.** Using `async_iter`, `async_map`, and `async_filter`, create a pipeline that generates the first 200 natural numbers, cubes each one, filters to keep only cubes less than 10,000, and prints the results.

5. **Priority Select.** Implement a "priority queue" using two channels: a high-priority channel and a low-priority channel. Write a consumer loop that uses `select` to prefer the high-priority channel but falls through to the low-priority one when the high-priority channel is empty. (Hint: use nested `select` with `default`.)

What's Next

You now have a solid grasp of channels—creation, send/recv semantics, closing, `select` multiplexing, timeouts, and async iterators. In the next chapter, we will step back and look at the bigger picture: when to use channels versus `Ref` for shared state, how to avoid deadlocks, and a collection of battle-tested concurrency patterns including worker pools, broadcast, and request-reply.

Chapter 21

Concurrency Patterns and Pitfalls

You now know the building blocks: `scope` and `spawn` for structured concurrency, `Channel` for message passing, `select` for multiplexing, and `Ref` for shared mutable state. But knowing the primitives and knowing how to *combine* them well are different skills. This chapter is a field guide—a collection of patterns that work, pitfalls that bite, and rules of thumb earned through experience.

21.1 `Ref` for Shared Mutable State

Before we compare `Ref` with channels, let's nail down exactly what `Ref` offers. A `Ref` is a reference-counted, heap-allocated wrapper around any Lattice value. Unlike ordinary values, which are deep-cloned when passed to spawns, a `Ref` is *shared by reference*—all threads see the same underlying data.

21.1.1 `Ref::new()`, `.get()`, `.set()`

Listing 21.1: Ref basics

```
let counter = Ref::new(0)

print(counter.get())      // 0
counter.set(42)
print(counter.get())      // 42
print(counter.inner_type()) // Int
```

The API is deliberately minimal:

- `Ref::new(value)` — Creates a new `Ref` wrapping the given value. Registered as a native function (`native_ref_new` in `src/runtime.c`).
- `.get()` (or `.deref()`) — Returns a deep clone of the inner value.
- `.set(value)` — Replaces the inner value. Fails on a frozen `Ref`.
- `.inner_type()` — Returns a string describing the type of the wrapped value.

Under the hood (see `src/stackvm.c`, the `VAL_REF` case in the builtin method handler), `.get()` calls `value_deep_clone` on the inner value, giving each caller its own independent copy. The `Ref` itself is a `LatRef` struct (`include/value.h`) containing a `LatValue` and a reference count.

21.1.2 Ref Proxying

A `Ref` proxies many methods to its inner value. If the inner value is an `Array`, you can call `.push()`, `.pop()`, and `.len()` directly on the `Ref`. If it is a `Map`, you can call `.get(key)`, `.set(key, val)`, `.keys()`, and more:

Listing 21.2: Ref proxying to inner types

```
let scores = Ref::new(Map::new())

// These calls go through to the inner Map:
scores.set("Alice", 95)
scores.set("Bob", 87)
print(scores.get("Alice"))  // 95
print(scores.len())         // 2
print(scores.keys())        // ["Alice", "Bob"]
```

Listing 21.3: Ref with Array inner type

```
let items = Ref::new([])

items.push("hammer")
items.push("nails")
items.push("screwdriver")
print(items.len())    // 3
print(items.get())    // ["hammer", "nails", "screwdriver"]
```

This proxying makes `Ref` feel natural—you rarely need to unwrap the inner value manually.

Freezing a Ref

You can `freeze` a `Ref`, which prevents `.set()`, `.push()`, `.pop()`, and other mutating operations. The inner value itself is not frozen—only the `Ref`'s ability to be mutated through its API is locked. This is useful when you want to share read-only data without sublimating.

21.2 When to Use Channels vs. `Ref`

This is one of the most common questions in concurrent Lattice programming: "Should I use a channel or a Ref?"

The short answer: **prefer channels for communication, use Ref for shared state that does not need message-passing semantics.**

Here is a decision guide:

Scenario	Channel	Ref
Passing results from producer to consumer		
Streaming data through a pipeline		
Shared configuration (read-mostly)		
Aggregate counter (write from many threads)		
Cancellation flag (boolean signal)		
Request-reply between two threads		
Accumulating results into a collection		*

Table 21.1: Channel vs. Ref decision matrix. *Ref works for accumulation but beware race conditions.

21.2.1 The Core Trade-Off

- **Channels** provide *ordering* and *blocking*. A `recv()` naturally waits for data, so you get synchronization for free. Channels also guarantee that each message is consumed exactly once.

- **Ref** provides *latest-value semantics*. A `.get()` always returns the current value, even if it has been overwritten multiple times. There is no ordering of writes, and no blocking—you always get an answer immediately.

Listing 21.4: Channel: ordered, exactly-once delivery

```
let ch = Channel::new()
ch.send(1)
ch.send(2)
ch.send(3)
print(ch.recv())  // 1 (always)
print(ch.recv())  // 2 (always)
print(ch.recv())  // 3 (always)
```

Listing 21.5: Ref: latest-value semantics

```
let state = Ref::new("initial")

scope {
  spawn { state.set("updated by A") }
  spawn { state.set("updated by B") }
}

print(state.get())  // "updated by A" or "updated by B"
```

Rule of Thumb

If you find yourself writing `ref.get()` followed by `ref.set(...)` in a spawn, you probably have a race condition. Restructure to use a channel: send partial results to an aggregator that does the read-modify-write on a single thread.

21.3 Deadlock Avoidance

A deadlock occurs when two or more threads are waiting for each other, and none can make progress. Lattice's structured concurrency model eliminates some deadlock patterns by design, but it does not eliminate all of them.

21.3.1 Classic Channel Deadlock

The most common deadlock in channel-based code: two threads each waiting to receive from the other's channel.

Listing 21.6: Deadlock: circular recv

```
// WARNING: This will deadlock!
let ch_a = Channel::new()
let ch_b = Channel::new()

scope {
  spawn {
    let val = ch_b.recv()  // Waiting for B to send
    ch_a.send(val + 1)
  }
  spawn {
    let val = ch_a.recv()  // Waiting for A to send
    ch_b.send(val + 1)
  }
}
// Neither spawn can proceed: both are blocked on recv.
```

Deadlock Is Not Detected

Lattice does not have a deadlock detector. If your program deadlocks, it will hang silently. The structured concurrency model means the `scope` will never return, and the program will appear frozen.

21.3.2 Rules for Avoiding Deadlocks

1. **Avoid circular channel dependencies.** If thread A needs a value from thread B and vice versa, redesign so that data flows in one direction, or use a mediator thread.
2. **Always close channels when done.** An unclosed channel causes `recv()` to block forever when the buffer is drained. A forgotten `close()` is the most common cause of "my program hangs" bugs.
3. **Use `select` with `timeout` for defensive waiting.** If you are not certain a value will arrive, protect yourself with a timeout:

Listing 21.7: Timeout as deadlock prevention

```
let result = select {
  slow_channel -> val { val }
  timeout 5000 { print("Possible deadlock!"); nil }
}
```

4. **Prefer fan-in to circular patterns.** Instead of two threads exchanging messages directly, have both send to a central coordinator channel and let a single thread make decisions.

5. **Send before receive when possible.** If a thread must both send and receive, do the sends first (or in a non-blocking way) so that you are not holding a blocking receive while another thread waits on your send.

21.3.3 Deadlock-Free Pattern: The Coordinator

Listing 21.8: Coordinator pattern avoids circular waits

```
let requests = Channel::new()
let responses = Channel::new()

scope {
  // Worker A: send request, wait for response
  spawn {
    requests.send("compute_42")
    let answer = responses.recv()
    print("A got: ${answer}")
  }

  // Coordinator: receive request, send response
  spawn {
    let req = requests.recv()
    if req == "compute_42" {
      responses.send(42)
    }
    requests.close()
    responses.close()
  }
}
```

Data flows in one direction (worker → coordinator → worker), so there is no cycle.

21.4 Common Patterns

21.4.1 Worker Pool

A fixed set of workers that consume from a shared job queue:

Listing 21.9: Worker pool pattern

```
fn run_worker_pool(jobs: Channel, results: Channel, num_workers: Int) {
  scope {
    // Launch workers --- use nested spawns for dynamic count
    spawn {
      scope {
        spawn {
          flux job = jobs.recv()
          while job != nil {
            results.send("W1: ${job} done")
            job = jobs.recv()
          }
        }
        spawn {
          flux job = jobs.recv()
          while job != nil {
            results.send("W2: ${job} done")
            job = jobs.recv()
          }
        }
        spawn {
          flux job = jobs.recv()
          while job != nil {
            results.send("W3: ${job} done")
            job = jobs.recv()
          }
        }
      }
      results.close()
    }
  }
}

let jobs = Channel::new()
let results = Channel::new()

// Enqueue jobs
for i in 0..9 {
  jobs.send("task_${i}")
}
jobs.close()

// Run the pool
run_worker_pool(jobs, results, 3)

// Collect results
flux r = results.recv()
while r != nil {
  print(r)
```

The key insight: because multiple workers call `recv()` on the same `jobs` channel, work is distributed automatically. When the channel is closed and drained, all workers exit their loops.

21.4.2 Broadcast

Channels are point-to-point: each value is consumed by exactly one receiver. To broadcast a value to multiple listeners, you need multiple channels:

Listing 21.10: Broadcast pattern

```
fn broadcast(value: any, listeners: Array) {
  for listener in listeners {
    listener.send(value)
  }
}

let listener_a = Channel::new()
let listener_b = Channel::new()
let listener_c = Channel::new()
let listeners = [listener_a, listener_b, listener_c]

broadcast("system shutting down", listeners)

// Each listener gets the message independently
print(listener_a.recv())  // system shutting down
print(listener_b.recv())  // system shutting down
print(listener_c.recv())  // system shutting down
```

For a real system, you might wrap this in a `Broadcaster` struct that manages the listener list and handles subscriptions:

Listing 21.11: Broadcaster with subscribers

```
fn make_broadcaster() {
  let subscribers = Ref::new([])

  let subscribe = |ch: Channel| {
    let current = subscribers.get()
    current.push(ch)
    subscribers.set(current)
  }

  let publish = |message: any| {
    let subs = subscribers.get()
    for sub in subs {
      sub.send(message)
    }
  }

  return [subscribe, publish]
}

let fns = make_broadcaster()
let subscribe = fns[0]
let publish = fns[1]

let ch1 = Channel::new()
let ch2 = Channel::new()
subscribe(ch1)
subscribe(ch2)

publish("hello everyone")
print(ch1.recv())  // hello everyone
print(ch2.recv())  // hello everyone
```

21.4.3 Request-Reply

A common pattern in concurrent services: a client sends a request along with a "reply channel," and the server sends the response back on that channel.

Listing 21.12: Request-reply pattern

```
let server_inbox = Channel::new()

scope {
  // Server: process requests
  spawn {
    flux req = server_inbox.recv()
    while req != nil {
      let query = req[0]
      let reply_ch = req[1]
      if query == "ping" {
        reply_ch.send("pong")
      } else {
        reply_ch.send("unknown: ${query}")
      }
      req = server_inbox.recv()
    }
  }

  // Client: send request with reply channel
  spawn {
    let reply = Channel::new()
    server_inbox.send(["ping", reply])
    print("Server says: ${reply.recv()}")

    let reply2 = Channel::new()
    server_inbox.send(["hello", reply2])
    print("Server says: ${reply2.recv()}")

    server_inbox.close()
  }
}
// Output:
// Server says: pong
// Server says: unknown: hello
```

Each request bundles a fresh reply channel. The server reads the request, computes the answer, and sends it back on the client's reply channel. This pattern decouples clients from the server's internal state and allows multiple clients to interact with a single server concurrently.

21.5 Testing Concurrent Code

Concurrent code is notoriously hard to test. Bugs may appear only under specific timing conditions, making them intermittent and frustrating. Here are strategies that help.

21.5.1 Deterministic Tests with Channels

The best concurrent tests eliminate timing from the equation. Use channels as synchronization points to create a deterministic order of operations:

Listing 21.13: Deterministic concurrent test

```
fn test_producer_consumer() {
  let ch = Channel::new()
  let results = Channel::new()

  scope {
    spawn {
      ch.send(10)
      ch.send(20)
      ch.send(30)
      ch.close()
    }
    spawn {
      flux sum = 0
      flux val = ch.recv()
      while val != nil {
        sum = sum + val
        val = ch.recv()
      }
      results.send(sum)
    }
  }

  let total = results.recv()
  if total != 60 {
    print("FAIL: expected 60, got ${total}")
  } else {
    print("PASS: producer-consumer sum is correct")
  }
}

test_producer_consumer()
```

The channels enforce ordering: the consumer sees all three values before computing the sum. No sleep calls, no timing assumptions.

21.5.2 Stress Testing with Repeated Runs

Some concurrency bugs only manifest under contention. Run your test many times:

Listing 21.14: Stress test via repeated runs

```
fn test_concurrent_counter_stress() {
  flux failures = 0

  for trial in 0..100 {
    let results = Channel::new()

    scope {
      spawn { results.send(1) }
      spawn { results.send(1) }
      spawn { results.send(1) }
    }

    let total = results.recv() + results.recv() + results.recv()
    if total != 3 {
      failures = failures + 1
    }
  }

  if failures > 0 {
    print("FAIL: ${failures}/100 trials failed")
  } else {
    print("PASS: all 100 trials correct")
  }
}

test_concurrent_counter_stress()
```

21.5.3 Testing with Timeouts

Protect tests from deadlocks by using `select` with a timeout:

Listing 21.15: Test with timeout protection

```
fn test_with_timeout() {
  let result_ch = Channel::new()

  scope {
    spawn {
      // Code under test
      result_ch.send("computed_value")
    }
  }

  let outcome = select {
    result_ch -> val { val }
    timeout 2000 { "TIMEOUT" }
  }

  if outcome == "TIMEOUT" {
    print("FAIL: test timed out (possible deadlock)")
  } else if outcome == "computed_value" {
    print("PASS: got expected result")
  } else {
    print("FAIL: unexpected result: ${outcome}")
  }
}

test_with_timeout()
```

If the test deadlocks, the timeout arm fires after 2 seconds instead of hanging forever.

21.5.4 Testing Error Propagation

Verify that errors from spawns reach the parent:

Listing 21.16: Testing error propagation

```
fn test_spawn_error_propagation() {
  flux caught_error = false

  try {
    scope {
      spawn {
        // Deliberately cause an error
        let arr = [1, 2, 3]
        let bad = arr[99]  // Out of bounds
      }
    }
  } catch err {
    caught_error = true
  }

  if caught_error {
    print("PASS: error from spawn was caught")
  } else {
    print("FAIL: error was not propagated")
  }
}

test_spawn_error_propagation()
```

21.5.5 Rules of Thumb for Concurrent Tests

1. **Never use sleep for synchronization.** Use channels instead—they provide both data transfer and synchronization in one primitive.

2. **Test the contract, not the timing.** Assert on *what* values you receive, not *when* you receive them. Concurrent order is nondeterministic by design.

3. **Run stress tests in CI.** A test that passes once might fail under load. Run concurrent tests 100+ times in your continuous integration pipeline.

4. **Isolate concurrent tests.** Each test should create its own channels and spawns. Shared global state between tests is a recipe for flaky results.

5. **Use timeouts as safety nets.** Every concurrent test should have a maximum runtime to prevent CI pipelines from hanging.

The GC and Concurrency

Each spawned thread gets its own `DualHeap` and GC state, so garbage collection in one thread does not pause other threads. If you are writing stress tests that allocate heavily inside spawns, you can be confident that GC pauses are thread-local. See the test file `tests/test_gc_concurrent.lat` for examples of GC-heavy concurrent workloads.

21.6 Exercises

1. **Safe Counter.** Implement a thread-safe counter using a channel: one spawn acts as the "counter server" and processes increment/get requests sent by other spawns. Verify that the final count matches the expected total.

2. **Deadlock Detection.** Write a program that deliberately deadlocks (two spawns doing circular `recv()`). Then fix it using the coordinator pattern described in Section 21.3.

3. **Worker Pool Benchmark.** Create a worker pool with 4 workers that processes 100 jobs. Each job is a string to be reversed. Collect all results and verify that every job was processed exactly once.

4. **Broadcast System.** Implement a pub-sub system using channels: a publisher broadcasts messages to all subscribers, each subscriber counts the messages it receives, and at the end each subscriber reports its count. Verify all subscribers received the same number of messages.

What's Next

With structured concurrency, channels, select, and the patterns in this chapter, you have a complete toolkit for writing concurrent Lattice programs that are safe, composable, and testable. In the next part of the book, we shift gears from the language core to the standard library—starting with files, paths, and the filesystem in **??**.

Part VI

The Standard Library

Chapter 22

Files, Paths, and the Filesystem

Every program eventually needs to talk to the outside world, and for most programs, that world is the filesystem. Whether you are reading a configuration file at startup, writing a log, or processing a batch of images, Lattice gives you a set of focused, no-surprises functions that map cleanly onto the underlying operating system. There are no deep class hierarchies to learn and no "stream" abstractions to wire together—just functions that take paths and return values.

In this chapter we will start with the three workhorses of text file I/O: `read_file`, `write_file`, and `append_file`. From there we will explore the broader `fs` family of functions for querying and manipulating the filesystem, the `path` helpers for constructing and dissecting file paths, temporary files and directories, and finally the `Buffer` type for working with raw binary data.

22.1 read_file, write_file, append_file

Let's start with the most common operation in scripting: reading a file into a string.

Listing 22.1: Reading and printing a file

```
let contents = read_file("config.toml")
print(contents)
```

`read_file` takes a path as a `String` and returns the entire file contents as a `String`. If the file does not exist or cannot be opened, it returns `nil`. Under the hood, the implementation in `src/builtins.c` opens the file with `fopen`, seeks to the end to determine the length, allocates exactly enough memory, and reads the contents in one pass.

Encoding Assumption

`read_file` assumes the file contains valid UTF-8 text. If you need to read arbitrary binary data—an image, a compiled bytecode file, a protocol buffer—use `read_file_bytes` instead, which returns a `Buffer`. We cover Buffers in Section 22.5.

Writing is the mirror operation:

Listing 22.2: Writing a string to a file

```
let report = "Scan completed: 0 issues found.\n"
write_file("report.txt", report)
```

`write_file` creates the file if it does not exist, or *truncates* it if it does. It returns `true` on success and `false` on failure. This is a full overwrite—if you need to add lines to an existing file, use `append_file`:

Listing 22.3: Appending to a log file

```
fn log_event(message: String) {
    let timestamp = time_format(time_now(), "%Y-%m-%d %H:%M:%S")
    let line = "[" + timestamp + "] " + message + "\n"
    append_file("app.log", line)
}

log_event("Server started on port 8080")
log_event("Accepted connection from 192.168.1.42")
```

`append_file` opens the file in append mode (`"a"` in C terms), so writes are guaranteed to go to the end of the file even if another process is writing concurrently. Like `write_file`, it creates the file if it does not exist.

write_file Truncates!

A common mistake is to use `write_file` when you meant `append_file`. If your program writes a log file using `write_file` in a loop, each iteration will obliterate everything the previous iteration wrote. When in doubt, ask yourself: "Do I want to replace or extend?"

22.1.1 A Complete Read-Modify-Write Cycle

A typical pattern combines all three operations. Suppose we have a simple counter file that stores a single integer:

Listing 22.4: Incrementing a persistent counter

```
fn increment_counter(path: String) -> Int {
    flux count = 0

    if file_exists(path) {
        let text = read_file(path)
        count = parse_int(text.trim())
    }

    count += 1
    write_file(path, to_string(count))
    return count
}

let visits = increment_counter("/tmp/visit_count.txt")
print("Visit #" + to_string(visits))
```

Notice how we check `file_exists` before reading. This is a deliberate choice—`read_file` returns `nil` on a missing file, but checking explicitly makes the intent clearer and lets us set a sensible default.

Prefer try/catch for Robust Code

For production code, wrap file operations in `try`/`catch` blocks (see Chapter 10). The examples in this chapter omit error handling for clarity, but real programs should always account for missing files, permission errors, and full disks.

22.2 The fs Module

Beyond reading and writing file contents, Lattice exposes a family of functions for querying and manipulating the filesystem itself. These are registered as top-level built-in functions—no import is required.

22.2.1 Checking Existence: file_exists, is_file, is_dir

Listing 22.5: Checking existence and type

```
let path = "/etc/hosts"

print(file_exists(path))   // true
print(is_file(path))       // true
print(is_dir(path))        // false
print(is_dir("/etc"))      // true
```

`file_exists` returns `true` if *anything* exists at the given path—file, directory, symlink, or device node. If you need to distinguish between files and directories, use `is_file` and `is_dir`. All three delegate to the POSIX `stat` system call internally (see `src/fs_ops.c`).

22.2.2 Listing Directories: list_dir

Listing 22.6: Listing directory entries

```
let entries = list_dir(".")
for entry in entries {
    print(entry)
}
```

`list_dir` returns an array of strings—one per entry in the directory, excluding the special `.` and `..` entries. The entries are file *names*, not full paths. If you need full paths, combine with `path_join`:

Listing 22.7: Building full paths from directory entries

```
let dir = "/var/log"
let entries = list_dir(dir)
for name in entries {
    let full_path = path_join(dir, name)
    let info = stat(full_path)
    print(full_path + " (" + info["type"] + ", " + to_string(info["size"]) + " bytes)")
}
```

22.2.3 Creating and Removing Directories: mkdir, rmdir

Listing 22.8: Creating and removing directories

```
mkdir("output")
write_file("output/results.csv", "name,score\nAlice,97\n")

// Later, clean up:
delete_file("output/results.csv")
rmdir("output")
```

`mkdir` creates a single directory with permissions `0755`. It does *not* create intermediate directories—if you need `mkdir -p` behavior, you will need to create each level yourself:

Listing 22.9: Creating nested directories

```
fn mkdir_p(path: String) {
    let parts = path.split("/")
    flux current = ""
    for part in parts {
        if part == "" {
            current = "/"
            continue
        }
        current = path_join(current, part)
        if !file_exists(current) {
            mkdir(current)
        }
    }
}

mkdir_p("output/reports/2024/q4")
```

`rmdir` removes an *empty* directory. Attempting to remove a non-empty directory will produce an error. If you need to recursively remove a directory tree, you must walk it yourself, deleting files first and directories bottom-up.

22.2.4 Globbing: glob

When you need to find files matching a pattern, `glob` is your friend:

Listing 22.10: Finding files with glob patterns

```
// Find all Lattice source files in the current directory
let sources = glob("*.lat")
for path in sources {
    print("Found: " + path)
}

// Find all JPEG images in a photos directory
let photos = glob("photos/*.jpg")
print("Found " + to_string(photos.len()) + " photos")
```

The pattern syntax follows the POSIX `glob(3)` convention: `*` matches any sequence of characters within a path component, `?` matches a single character, and `[abc]` matches character classes. On systems that support it, brace expansion (`{jpg,png,gif}`) is also available.

Listing 22.11: Brace expansion in glob patterns

```
// Find all image files (on systems with GLOB_BRACE support)
let images = glob("assets/*.{jpg,png,gif,webp}")
```

If no files match, `glob` returns an empty array—it does not treat the absence of matches as an error. You can see this behavior in `src/fs_ops.c`, where `GLOB_NOMATCH` returns `NULL` with a count of zero rather than setting the error string.

22.2.5 File Metadata: stat and file_size

The `stat` function returns a map with detailed metadata about a path:

Listing 22.12: Inspecting file metadata with stat

```
let info = stat("README.md")
print("Type: " + info["type"])          // "file"
print("Size: " + to_string(info["size"]) + " bytes")
print("Modified: " + to_string(info["mtime"]))  // epoch milliseconds
print("Mode: " + to_string(info["mode"]))       // permission bits
```

The returned map contains five keys:

Key	Type	Description
`"type"`	`String`	`"file"`, `"dir"`, `"symlink"`, or `"other"`
`"size"`	`Int`	Size in bytes
`"mtime"`	`Int`	Last modification time (epoch milliseconds)
`"mode"`	`Int`	Unix permission bits (e.g., `0o755`)
`"permissions"`	`Int`	Same as `"mode"` (alias)

If you only need the file size, `file_size` is a convenient shortcut that returns an `Int` directly:

Listing 22.13: Quick file size check

```
let bytes = file_size("database.db")
let megabytes = bytes / (1024 * 1024)
print("Database is " + to_string(megabytes) + " MB")
```

22.2.6 Copying, Renaming, and Deleting

Listing 22.14: File operations: copy, rename, delete

```
// Make a backup before modifying
copy_file("config.toml", "config.toml.bak")

// Rename (move) a file
rename("draft.md", "final.md")

// Clean up temporary files
delete_file("scratch.tmp")
```

`copy_file` reads the source in 8 KB chunks and writes them to the destination, making it safe for large files. `rename` performs an atomic rename on the same filesystem (it delegates to the C `rename()` function). `delete_file` unlinks a single file; it cannot remove directories.

22.2.7 Permissions and Canonical Paths

Listing 22.15: Changing permissions and resolving paths

```
// Make a script executable
write_file("deploy.sh", "#!/bin/sh\necho 'Deploying...'\n")
chmod("deploy.sh", 0o755)

// Resolve symlinks and relative components
let canonical = realpath("./src/../README.md")
print(canonical)  // "/home/user/project/README.md"
```

`chmod` takes a path and an integer mode. You can use octal literals like `0o755` for readability. `realpath` resolves a path to its absolute, canonical form—expanding symlinks, removing `.` and `..` components, and verifying that the path exists.

Filesystem Functions at a Glance

All filesystem functions are registered as top-level built-ins in `src/runtime.c`. They require no imports. On Emscripten (browser) builds, all filesystem operations return errors, since there is no real filesystem available—check the Emscripten stubs in `src/fs_ops.c` if you are curious about the boundary.

22.3 The path Module

File paths are strings, but they have structure. The path functions help you manipulate that structure without resorting to brittle string concatenation.

22.3.1 Joining Paths: path_join

Listing 22.16: Joining path components

```
let dir = "/home/user/projects"
let file = "lattice/main.lat"
let full = path_join(dir, file)
print(full)  // "/home/user/projects/lattice/main.lat"
```

`path_join` is variadic—you can pass any number of string arguments:

Listing 22.17: Joining multiple path segments

```
let config_path = path_join("/etc", "lattice", "config.toml")
print(config_path)  // "/etc/lattice/config.toml"
```

The implementation in `src/path_ops.c` is careful about separators. It avoids producing double slashes (`//`) when one component ends with a slash and the next begins with one, and it inserts a separator when neither component has one. This means you do not have to worry about trailing slashes on your directory paths:

Listing 22.18: path_join handles trailing slashes

```
// All of these produce the same result:
print(path_join("/tmp/", "file.txt"))   // "/tmp/file.txt"
print(path_join("/tmp", "file.txt"))    // "/tmp/file.txt"
print(path_join("/tmp/", "/file.txt"))  // "/tmp/file.txt"
```

22.3.2 Dissecting Paths: path_dir, path_base, path_ext

Three functions let you take a path apart:

Listing 22.19: Dissecting a file path

```
let path = "/home/user/photos/vacation.jpg"

print(path_dir(path))   // "/home/user/photos"
print(path_base(path))  // "vacation.jpg"
print(path_ext(path))   // ".jpg"
```

`path_dir` returns the directory portion of a path—everything before the last slash. If the path has no slash, it returns `"."` (the current directory).

`path_base` returns the filename—everything after the last slash. If the path ends with a slash, it returns an empty string.

`path_ext` returns the file extension, *including the dot*. If there is no extension, or if the filename starts with a dot (like `.gitignore`), it returns an empty string.

Listing 22.20: Edge cases in path functions

```
// No directory component
print(path_dir("readme.md"))   // "."
print(path_base("readme.md"))  // "readme.md"

// Hidden files
print(path_ext(".gitignore"))  // "" (dot-prefixed names are not extensions)

// No extension
print(path_ext("Makefile"))    // ""

// Root path
print(path_dir("/"))           // "/"
```

22.3.3 Building a File Organizer

Let's combine path functions with filesystem operations to build something practical—a script that organizes files into directories by extension:

Listing 22.21: Organizing files by extension

```
fn organize_by_extension(source_dir: String) {
    let files = list_dir(source_dir)

    for name in files {
        let full_path = path_join(source_dir, name)

        // Skip directories
        if is_dir(full_path) { continue }

        // Determine the target directory
        let ext = path_ext(name)
        if ext == "" { ext = "no_extension" }
        // Remove the leading dot
        let category = ext.slice(1, ext.len())

        let target_dir = path_join(source_dir, category)
        if !file_exists(target_dir) {
            mkdir(target_dir)
        }

        let dest = path_join(target_dir, name)
        rename(full_path, dest)
        print("Moved " + name + " -> " + category + "/")
    }
}

organize_by_extension("/tmp/downloads")
```

Use path Functions, Not String Slicing

It is tempting to split paths on `"/"` manually, but `path_dir`, `path_base`, and `path_ext` handle edge cases that manual string manipulation easily gets wrong: root paths, trailing slashes, dot-prefixed filenames, and empty components. Always prefer the path functions.

22.4 Temporary Files and Directories

Sometimes you need a place to stash data that should not outlive your program. Lattice provides two functions for this: `tempfile` and `tempdir`.

Listing 22.22: Creating temporary files and directories

```
// Create a temporary file
let tmp_path = tempfile()
print(tmp_path)  // e.g., "/tmp/lattice_a3kJ9x"

write_file(tmp_path, "intermediate results go here")
let data = read_file(tmp_path)
print(data)  // "intermediate results go here"

// Clean up when done
delete_file(tmp_path)
```

`tempfile` creates a new file in the system's temporary directory (typically `/tmp` on Unix or the path returned by `GetTempPathA` on Windows) with a unique name based on the template `lattice_XXXXXX`. Internally, it uses `mkstemp`, which atomically creates the file and guarantees no name collisions. The file descriptor is immediately closed, and the path is returned as a string.

`tempdir` works the same way but creates a directory instead:

Listing 22.23: Using a temporary directory as a workspace

```
let workspace = tempdir()
print(workspace)  // e.g., "/tmp/lattice_Qm7kP2"

// Use the directory for intermediate files
write_file(path_join(workspace, "step1.txt"), "Phase 1 output")
write_file(path_join(workspace, "step2.txt"), "Phase 2 output")

// List what we created
let files = list_dir(workspace)
for name in files {
    print("  " + name)
}

// Clean up
for name in files {
    delete_file(path_join(workspace, name))
}
rmdir(workspace)
```

Temporary Files Are Not Automatically Cleaned Up

Lattice does not use finalizers or RAII-style destructors to clean up temporary files. If your program exits without deleting them, they will remain on disk until the operating system's periodic cleanup runs (if ever). Always delete your temporaries explicitly, ideally using a `defer` block to ensure cleanup happens even if an error occurs:

```
let tmp = tempfile()
defer { delete_file(tmp) }
// ... work with the file ...
```

22.4.1 A Safe Processing Pipeline

A robust pattern for file processing is to write output to a temporary file, then atomically rename it to the final destination. This prevents other programs from reading a half-written file:

Listing 22.24: Atomic file writes with tempfile and rename

```
fn safe_write(destination: String, content: String) {
    let dir = path_dir(destination)
    let tmp = tempfile()
    defer {
        // Clean up the temp file if rename failed
        if file_exists(tmp) {
            delete_file(tmp)
        }
    }

    write_file(tmp, content)
    rename(tmp, destination)
}

safe_write("output/report.json", "{\"status\": \"complete\"}")
```

The `defer` block ensures the temporary file is cleaned up no matter what happens. If the `rename` succeeds, the temporary file no longer exists at its original path, so the `file_exists` check prevents a spurious error.

22.5 Working with Buffers

Strings are great for text, but the world is full of binary data: images, network packets, serialized structures, compressed archives. Lattice represents raw byte sequences with the `Buffer` type.

Buffer

A *Buffer* is a contiguous, mutable sequence of bytes (`uint8_t` values, ranging from 0 to 255). Buffers have a length (the number of bytes currently stored) and a capacity (the allocated space). You can index into a Buffer to read or write individual bytes, and use typed read/write methods to interpret sequences of bytes as integers or floating-point values.

22.5.1 Creating Buffers

There are several ways to create a Buffer:

Listing 22.25: Creating Buffers

```
// Allocate a zero-filled buffer of a given size
let zeros = Buffer::new(16)
print(zeros.len())  // 16

// Create from an array of byte values
let header = Buffer::from([0x89, 0x50, 0x4E, 0x47])
print(header.len())  // 4

// Create from a string (UTF-8 bytes)
let greeting = Buffer::from_string("Hello")
print(greeting.len())  // 5
print(greeting[0])     // 72 (ASCII 'H')
```

`Buffer::new(size)` allocates a buffer of the given size, filled with zeros. `Buffer::from(array)` creates a buffer from an array of integers, truncating each value to the range 0–255. `Buffer::from_string(str)` copies the UTF-8 bytes of a string into a new buffer.

22.5.2 Reading and Writing Binary Files

To read a file as raw bytes, use `read_file_bytes`:

Listing 22.26: Reading a binary file

```
let data = read_file_bytes("image.png")
print("File size: " + to_string(data.len()) + " bytes")

// Check the PNG magic number
if data[0] == 0x89 && data[1] == 0x50 && data[2] == 0x4E && data[3] == 0x47 {
    print("Valid PNG header!")
}
```

To write a buffer to disk, use `write_file_bytes`:

Listing 22.27: Writing binary data to a file

```
// Create a minimal BMP file header (simplified)
flux bmp = Buffer::new(0)
bmp.push(0x42)  // 'B'
bmp.push(0x4D)  // 'M'

// Write to disk
write_file_bytes("output.bmp", bmp)
```

22.5.3 Indexing and Mutation

Buffers support integer indexing for both reading and writing:

Listing 22.28: Buffer indexing

```
flux buf = Buffer::new(4)
buf[0] = 0xFF
buf[1] = 0xCA
buf[2] = 0xFE
buf[3] = 0x00

print(buf[0])  // 255
print(buf[2])  // 254

// Values are masked to a single byte
buf[0] = 256
print(buf[0])  // 0 (only the low 8 bits are kept)
```

When you assign a value to a buffer index, Lattice masks it with `0xFF`, keeping only the lowest 8 bits. Reading an index returns an `Int` in the range 0–255. Out-of-bounds access produces a runtime error.

22.5.4 The push Method and Dynamic Growth

Buffers grow dynamically when you use `push`:

Listing 22.29: Building a buffer byte by byte

```
flux packet = Buffer::new(0)
packet.push(0x01)      // version
packet.push(0x00)      // flags
packet.push_u16(1024)  // payload length as 16-bit value
packet.push_u32(42)    // sequence number as 32-bit value

print(packet.len())       // 8
print(packet.capacity())  // at least 8
```

`push` appends a single byte. `push_u16` and `push_u32` append multi-byte values in little-endian byte order.

22.5.5 Typed Read and Write Methods

When working with structured binary data, you need to read and write multi-byte values at specific offsets. Buffers provide a full suite of typed accessors:

Listing 22.30: Typed buffer reads and writes

```
flux buf = Buffer::new(20)

// Write values at specific offsets
buf.write_u8(0, 0xFF)        // 1 byte at offset 0
buf.write_u16(1, 0xCAFE)     // 2 bytes at offset 1
buf.write_u32(3, 0xDEADBEEF) // 4 bytes at offset 3

// Read them back
print(buf.read_u8(0))   // 255
print(buf.read_u16(1))  // 51966  (0xCAFE)
print(buf.read_u32(3))  // 3735928559  (0xDEADBEEF)
```

The full set of typed methods is:

Read	Write	Description
`read_u8(offset)`	`write_u8(offset, val)`	Unsigned 8-bit integer (1 byte)
`read_u16(offset)`	`write_u16(offset, val)`	Unsigned 16-bit integer (2 bytes)
`read_u32(offset)`	`write_u32(offset, val)`	Unsigned 32-bit integer (4 bytes)
`read_u64(offset)`	`write_u64(offset, val)`	Unsigned 64-bit integer (8 bytes)
`read_i8(offset)`	—	Signed 8-bit integer
`read_i16(offset)`	—	Signed 16-bit integer
`read_i32(offset)`	—	Signed 32-bit integer
`read_i64(offset)`	`write_i64(offset, val)`	Signed 64-bit integer
`read_f32(offset)`	—	IEEE 754 single-precision float (4 bytes)
`read_f64(offset)`	—	IEEE 754 double-precision float (8 bytes)

All multi-byte reads and writes use *little-endian* byte order, matching the native byte order on x86 and ARM systems.

Bounds Checking

All typed read and write methods perform bounds checking. Reading a `u32` at offset 5 from a 7-byte buffer will produce a runtime error, because the read would extend past the end of the buffer (offset 5 + 4 bytes = offset 9, which exceeds length 7). Always verify your offsets when parsing binary formats.

22.5.6 Slicing, Filling, and Converting

Buffers provide several utility methods for common operations:

Listing 22.31: Buffer utility methods

```
let data = Buffer::from_string("Hello, Lattice!")

// Slice: extract a sub-buffer (start, end)
let sub = data.slice(0, 5)
print(sub.to_string())  // "Hello"

// to_hex: get a hex string representation
let header = Buffer::from([0xDE, 0xAD, 0xBE, 0xEF])
print(header.to_hex())  // "deadbeef"

// to_array: convert to an array of integers
let bytes = header.to_array()
print(bytes)  // [222, 173, 190, 239]

// fill: set all bytes to a value
flux scratch = Buffer::new(8)
scratch.fill(0xFF)
print(scratch[3])  // 255

// clear: reset length to zero
scratch.clear()
print(scratch.len())  // 0

// resize: change the buffer length
flux resizable = Buffer::new(4)
resizable.resize(8)
print(resizable.len())  // 8
```

`slice(start, end)` returns a *new* buffer containing the bytes from index `start` up to (but not including) end. `to_string` interprets the buffer bytes as UTF-8 and returns a `String`. `to_hex` produces a lowercase hexadecimal representation. `to_array` returns an `Array` of `Int` values, one per byte.

22.5.7 Parsing a Binary File Format

Let's put everything together by parsing a WAV audio file header. The WAV format starts with a well-defined 44-byte header:

Listing 22.32: Parsing a WAV file header

```
fn parse_wav_header(path: String) {
    let data = read_file_bytes(path)
    if data.len() < 44 {
        print("File too small to be a valid WAV")
        return
    }

    // Bytes 0-3: "RIFF" marker
    let riff = data.slice(0, 4).to_string()
    if riff != "RIFF" {
        print("Not a RIFF file")
        return
    }

    // Bytes 4-7: file size minus 8
    let file_size = data.read_u32(4) + 8

    // Bytes 8-11: "WAVE" format
    let format = data.slice(8, 12).to_string()
    if format != "WAVE" {
        print("Not a WAVE file")
        return
    }

    // Bytes 22-23: number of channels
    let channels = data.read_u16(22)

    // Bytes 24-27: sample rate
    let sample_rate = data.read_u32(24)

    // Bytes 34-35: bits per sample
    let bits_per_sample = data.read_u16(34)

    print("WAV File: " + path)
    print("  Channels:       " + to_string(channels))
    print("  Sample rate:    " + to_string(sample_rate) + " Hz")
    print("  Bits per sample: " + to_string(bits_per_sample))
    print("  File size:      " + to_string(file_size) + " bytes")
}

parse_wav_header("recording.wav")
```

This example demonstrates the core workflow for binary file parsing: read the file as bytes, check magic numbers, and use typed reads at known offsets to extract structured fields.

Buffers Are Value Types

Like structs in Lattice, Buffers are *value types*—assigning a buffer to a new variable or passing it to a function creates a copy. If you modify the copy, the original is unaffected. This eliminates a whole category of aliasing bugs but means you should be mindful of copying large buffers unnecessarily. For more on value semantics, see Chapter 3.

Exercises

1. **Disk Usage Counter.** Write a function `disk_usage(dir: String) -> Int` that recursively calculates the total size in bytes of all files in a directory tree. Use `list_dir`, `is_dir`, `is_file`, and `file_size`.

2. **Find and Replace.** Write a program that takes a directory path, a search string, and a replacement string as arguments. It should find all `.txt` files in the directory using `glob`, read each one, replace all occurrences of the search string, and write the result back. Use the safe-write pattern from Section 22.4.1 to avoid corrupting files.

3. **Hex Dump.** Write a function `hex_dump(path: String, count: Int)` that reads the first `count` bytes of a file using `read_file_bytes` and prints them in a classic hex dump format: 16 bytes per line, showing both hexadecimal values and printable ASCII characters.

4. **File Deduplication.** Write a program that scans a directory, reads every file into a Buffer, computes a checksum (you can use the `to_hex` method on the bytes for a simple hash, or wait until Chapter 25 for `sha256`), and reports any duplicate files.

5. **Binary Concatenator.** Write a function that takes an array of file paths and a destination path, reads each file as bytes, and concatenates all the buffers into a single file. Pay attention to building the output buffer efficiently.

What's Next

Now that we can read and write files, the natural next question is: what *format* should those files be in? In the next chapter, we will explore Lattice's built-in support for four of the most common data formats—JSON, TOML, YAML, and CSV—and see how to parse, generate, and round-trip structured data between them.

Chapter 23

JSON, TOML, YAML, and CSV

Programs rarely exist in isolation. They read configuration files, consume API responses, export reports, and exchange data with other programs. The format of that data matters, and there is no single format that fits every situation. JSON dominates web APIs. TOML is the configuration language of choice for many modern tools. YAML appears in CI pipelines and Kubernetes manifests. CSV is the lingua franca of spreadsheets and data science.

Lattice provides built-in support for all four. Each format has a `_parse` function that turns a string into Lattice values, and a `_stringify` function that turns Lattice values back into a string. The mapping between formats and Lattice types is consistent and predictable: JSON objects become Maps, arrays become Arrays, and scalars become the corresponding Lattice primitives.

Let's start with JSON, the format you will probably use most often.

23.1 json_parse / json_stringify

23.1.1 Parsing JSON

Listing 23.1: Parsing a JSON string

```
let text = "{\"name\": \"Lattice\", \"version\": 3, \"stable\": true}"
let data = json_parse(text)

print(data["name"])     // "Lattice"
print(data["version"])  // 3
print(data["stable"])   // true
```

`json_parse` takes a `String` and returns a Lattice value. The type mapping follows the JSON specification exactly:

JSON Type	Lattice Type
object	`Map`
array	`Array`
string	`String`
number (integer)	`Int`
number (decimal/exponent)	`Float`
`true` / `false`	`Bool`
`null`	`nil`

Notice that the parser distinguishes between integers and floats: `42` becomes an `Int`, while `42.0` or `4.2e1` becomes a `Float`. This matters because integer arithmetic in Lattice is exact, while floating-point arithmetic is subject to the usual IEEE 754 rounding.

Listing 23.2: Parsing nested JSON structures

```
let json_text = """
{
    "users": [
        {"id": 1, "name": "Alice", "active": true},
        {"id": 2, "name": "Bob", "active": false}
    ],
    "count": 2
}
"""

let data = json_parse(json_text)
let users = data["users"]

for user in users {
    let status = if user["active"] { "active" } else { "inactive" }
    print(user["name"] + " is " + status)
}
// Alice is active
// Bob is inactive
```

Error Handling in json_parse

If the input is not valid JSON, `json_parse` sets a runtime error with a message indicating the position of the problem. The parser in `src/json.c` uses a recursive descent approach and validates strictly—trailing content after the root value is rejected, and all escape sequences must be well-formed. Wrap parsing calls in `try`/`catch` if you are dealing with untrusted input.

23.1.2 Stringifying to JSON

Listing 23.3: Converting Lattice values to JSON

```
let user = Map::new()
user["name"] = "Charlie"
user["age"] = 28
user["hobbies"] = ["climbing", "cooking", "Lattice"]

let json_text = json_stringify(user)
print(json_text)
// {"name":"Charlie","age":28,"hobbies":["climbing","cooking","Lattice"]}
```

`json_stringify` accepts any Lattice value that has a natural JSON representation: Maps, Arrays, Strings, Ints, Floats, Bools, and `nil`. Tuples and Buffers are serialized as JSON arrays. Refs are unwrapped automatically.

Unsupported Types

Structs, closures, channels, ranges, sets, iterators, and enums cannot be serialized to JSON directly. Attempting to stringify one of these will produce a runtime error. If you need to serialize a struct, convert it to a Map first.

The serializer handles special floating-point values gracefully: `Infinity`, `-Infinity`, and `NaN` are all encoded as `null`, since JSON has no representation for them. Strings are properly escaped, including Unicode sequences (`\uXXXX`) for control characters below `U+0020`.

23.1.3 A Practical Example: Reading and Writing JSON Files

Listing 23.4: Loading and saving application state as JSON

```
fn load_state(path: String) -> Map {
    if !file_exists(path) {
        let defaults = Map::new()
        defaults["theme"] = "dark"
        defaults["font_size"] = 14
        defaults["recent_files"] = []
        return defaults
    }
    let text = read_file(path)
    return json_parse(text)
}

fn save_state(path: String, state: Map) {
    let text = json_stringify(state)
    write_file(path, text)
}

// Load, modify, save
flux state = load_state("settings.json")
state["font_size"] = 16
state["recent_files"] = ["main.lat", "lib/utils.lat"]
save_state("settings.json", state)
```

23.2 toml_parse / toml_stringify

TOML (Tom's Obvious Minimal Language) was designed to be a configuration file format that is easy to read and write by hand. If JSON is for machines, TOML is for humans.

23.2.1 Parsing TOML

Listing 23.5: Parsing a TOML configuration

```
let toml_text = """
title = "My Project"
version = 3

[database]
host = "localhost"
port = 5432
enabled = true

[logging]
level = "info"
file = "/var/log/app.log"
"""

let config = toml_parse(toml_text)
print(config["title"])              // "My Project"
print(config["database"]["host"])   // "localhost"
print(config["database"]["port"])   // 5432
print(config["logging"]["level"])   // "info"
```

Like `json_parse`, the TOML parser returns a Map. Table headers (`[section]`) create nested Maps. The parser supports all standard TOML features:

- Bare keys (`name = "value"`) and quoted keys (`"complex key" = "value"`)
- Dotted keys (`server.host = "localhost"`) that create nested maps automatically
- Single-quoted strings (literal, no escape processing) and double-quoted strings (with escape sequences)
- Integers (including underscored: `1_000_000`), floats, and booleans
- Inline arrays (`[1, 2, 3]`) and inline tables (`{key = "value"}`)
- Array-of-tables syntax (`[[items]]`)

Listing 23.6: TOML array of tables

```
let manifest = """
[package]
name = "my-app"
version = "1.0.0"

[[dependencies]]
name = "http"
version = "0.5.0"

[[dependencies]]
name = "json"
version = "1.2.0"
"""

let pkg = toml_parse(manifest)
print(pkg["package"]["name"])  // "my-app"

let deps = pkg["dependencies"]
for dep in deps {
    print(dep["name"] + " v" + dep["version"])
}
// http v0.5.0
// json v1.2.0
```

Each `[[dependencies]]` block pushes a new Map onto an array under the `"dependencies"` key. This is how TOML represents arrays of structured data.

23.2.2 Stringifying to TOML

Listing 23.7: Generating TOML from Lattice values

```
let config = Map::new()
config["title"] = "Generated Config"
config["debug"] = false

let server = Map::new()
server["host"] = "0.0.0.0"
server["port"] = 8080
config["server"] = server

let output = toml_stringify(config)
print(output)
// title = "Generated Config"
// debug = false
//
// [server]
// host = "0.0.0.0"
// port = 8080
```

The TOML serializer in `src/toml_ops.c` uses a two-pass approach: first it emits all scalar key-value pairs for the current table, then it recurses into sub-tables and arrays of tables. This produces clean, well-structured TOML output.

toml_stringify Requires a Map

Unlike `json_stringify`, which accepts any serializable value, `toml_stringify` requires its argument to be a Map. TOML documents are always rooted in a table (a Map), so passing an Array or a String will produce a runtime error.

23.3 yaml_parse / yaml_stringify

YAML is the most flexible of the four formats. It uses indentation for structure (like Python), supports both block and flow styles, and has extensive scalar type detection.

23.3.1 Parsing YAML

Listing 23.8: Parsing a YAML document

```
let yaml_text = """
name: Lattice
version: 3
features:
  - phases
  - structured concurrency
  - pattern matching
database:
  host: localhost
  port: 5432
  ssl: true
"""

let data = yaml_parse(yaml_text)
print(data["name"])               // "Lattice"
print(data["database"]["host"])   // "localhost"
print(data["database"]["ssl"])    // true

for feature in data["features"] {
    print("- " + feature)
}
// - phases
// - structured concurrency
// - pattern matching
```

Lattice's YAML parser handles the most commonly used subset of YAML:

- Block mappings (`key: value`)
- Block sequences (`- item`)
- Flow sequences (`[a, b, c]`) and flow mappings (`{a: 1, b: 2}`)
- Quoted and unquoted strings
- Document separators (`---`)

Automatic Scalar Detection

The YAML parser automatically detects scalar types from unquoted values. The words `true`, `True`, `yes`, and `Yes` all become `Bool true`. Similarly, `false`, `no`, and their capitalized variants become `false`. The words `null`, `Null`, and `~` become `nil`. Numeric-looking strings are parsed as `Int` or `Float`. If none of these patterns match, the value remains a `String`. You can see this logic in the `yaml_detect_scalar` function in `src/yaml_ops.c`.

Listing 23.9: YAML scalar auto-detection

```
let yaml_text = """
string_val: hello
int_val: 42
float_val: 3.14
bool_val: true
null_val: null
yes_val: yes
"""

let data = yaml_parse(yaml_text)
print(typeof(data["string_val"]))  // "String"
print(typeof(data["int_val"]))     // "Int"
print(typeof(data["float_val"]))   // "Float"
print(typeof(data["bool_val"]))    // "Bool"
print(typeof(data["null_val"]))    // "Nil"
print(typeof(data["yes_val"]))     // "Bool"
```

Force Strings with Quotes

If you need a value like `yes` or `3.14` to remain a string rather than being auto-detected as a boolean or float, wrap it in quotes in your YAML source: `value: "yes"` or `pi: "3.14"`.

23.3.2 Stringifying to YAML

Listing 23.10: Generating YAML from Lattice values

```
let deployment = Map::new()
deployment["apiVersion"] = "apps/v1"
deployment["kind"] = "Deployment"

let metadata = Map::new()
metadata["name"] = "my-app"
metadata["labels"] = Map::new()
metadata["labels"]["app"] = "my-app"
deployment["metadata"] = metadata

let yaml_out = yaml_stringify(deployment)
print(yaml_out)
// apiVersion: apps/v1
// kind: Deployment
// metadata:
//   name: my-app
//   labels:
//     app: my-app
```

The YAML serializer produces clean, indented block-style output. It automatically quotes strings that could be misinterpreted as YAML special values—so a string containing `"true"` will be emitted as `"true"` rather than bare `true`. Empty maps and arrays use the compact flow forms `{}` and `[]`.

`yaml_stringify` accepts Maps and Arrays. Other types produce a runtime error.

23.4 csv_parse / csv_stringify

CSV is the oldest and most universal data exchange format. It has no formal specification (though RFC 4180 comes close), but the basic idea is universal: rows separated by newlines, fields separated by commas.

23.4.1 Parsing CSV

Listing 23.11: Parsing CSV data

```
let csv_text = "name,age,city\nAlice,30,Portland\nBob,25,Seattle\n"
let rows = csv_parse(csv_text)

for row in rows {
    print(row)
}
// ["name", "age", "city"]
// ["Alice", "30", "Portland"]
// ["Bob", "25", "Seattle"]
```

`csv_parse` returns an Array of Arrays—each inner array represents one row, and each element is a `String`. Unlike JSON and YAML, CSV has no type system, so all values come back as strings. If you need numbers, use `parse_int` or `parse_float` on the individual fields.

The parser handles the important edge cases:

- **Quoted fields**: Fields enclosed in double quotes can contain commas, newlines, and other special characters.
- **Escaped quotes**: A double quote inside a quoted field is escaped by doubling it: `"He said ""hello"""` produces `He said "hello"`.
- **Windows line endings**: Both `\n` and `\r\n` are handled.

Listing 23.12: Parsing CSV with quoted fields

```
let csv_text = """
name,bio,score
Alice,"Enjoys hiking, reading",95
Bob,"Said ""hello"" to everyone",87
"""

let rows = csv_parse(csv_text)
print(rows[1][1])  // "Enjoys hiking, reading"
print(rows[2][1])  // "Said \"hello\" to everyone"
```

23.4.2 Working with Headers

Since CSV does not distinguish between header rows and data rows, you will often want to convert the raw arrays into a more convenient structure:

Listing 23.13: Converting CSV rows to maps

```
fn csv_to_maps(rows: Array) -> Array {
    let headers = rows[0]
    flux results = []

    flux i = 1
    while i < rows.len() {
        let row = rows[i]
        let record = Map::new()
        flux j = 0
        while j < headers.len() {
            if j < row.len() {
                record[headers[j]] = row[j]
            }
            j += 1
        }
        results.push(record)
        i += 1
    }
    return results
}

let csv_text = "name,age,city\nAlice,30,Portland\nBob,25,Seattle\n"
let records = csv_to_maps(csv_parse(csv_text))

for record in records {
    print(record["name"] + " lives in " + record["city"])
}
// Alice lives in Portland
// Bob lives in Seattle
```

23.4.3 Stringifying to CSV

Listing 23.14: Generating CSV output

```
let data = [
    ["product", "price", "quantity"],
    ["Widget", "9.99", "100"],
    ["Gadget", "24.99", "50"],
    ["Doohickey", "4.99", "200"]
]

let csv_text = csv_stringify(data)
print(csv_text)
// product,price,quantity
// Widget,9.99,100
// Gadget,24.99,50
// Doohickey,4.99,200
```

`csv_stringify` takes an Array of Arrays and produces a CSV string. Each inner array becomes a row. Fields that contain commas, double quotes, or newlines are automatically wrapped in double quotes, with internal quotes escaped by doubling.

Listing 23.15: CSV auto-quoting of special characters

```
let data = [
    ["name", "description"],
    ["Widget", "A small, useful device"],
    ["Gadget", "Says \"beep\" loudly"]
]

let csv_text = csv_stringify(data)
print(csv_text)
// name,description
// Widget,"A small, useful device"
// Gadget,"Says ""beep"" loudly"
```

23.5 Round-Tripping Data Between Formats

One of the advantages of Lattice's consistent parsing model is that data flows naturally between formats. Parse from one format, stringify to another. The intermediate representation—Maps, Arrays, and primitives—is the same regardless of which format you started with.

Listing 23.16: Converting TOML configuration to JSON

```
let toml_text = read_file("config.toml")
let config = toml_parse(toml_text)

// Convert to JSON for an API or another tool
let json_text = json_stringify(config)
write_file("config.json", json_text)
print("Converted TOML to JSON")
```

Listing 23.17: Converting YAML to TOML

```
let yaml_text = """
server:
  host: 0.0.0.0
  port: 8080
  workers: 4
database:
  url: postgres://localhost/mydb
  pool_size: 10
"""

let config = yaml_parse(yaml_text)
let toml_text = toml_stringify(config)
write_file("config.toml", toml_text)
print(toml_text)
// [server]
// host = "0.0.0.0"
// port = 8080
// workers = 4
//
// [database]
// url = "postgres://localhost/mydb"
// pool_size = 10
```

23.5.1 CSV to JSON

CSV-to-JSON conversion is particularly common when preparing data for web APIs:

Listing 23.18: Converting CSV to JSON

```
let csv_text = read_file("employees.csv")
let rows = csv_parse(csv_text)

// First row is headers
let headers = rows[0]
flux records = []

flux i = 1
while i < rows.len() {
    let row = rows[i]
    let record = Map::new()
    flux j = 0
    while j < headers.len() {
        let field = if j < row.len() { row[j] } else { "" }
        // Try to parse numeric fields
        if headers[j] == "age" || headers[j] == "salary" {
            record[headers[j]] = parse_int(field)
        } else {
            record[headers[j]] = field
        }
        j += 1
    }
    records.push(record)
    i += 1
}

let json_text = json_stringify(records)
write_file("employees.json", json_text)
```

Lossy Round-Trips

Not all round-trips are lossless. TOML has no `null` type, so a JSON `null` will be lost or turned into an empty string when converted to TOML. YAML's auto-detection means that the string `"true"` in JSON might become a boolean if you round-trip through YAML without quoting. CSV loses all type information entirely—everything becomes strings. Be aware of these edge cases when converting between formats.

23.6 Building Configuration-Driven Programs

Let's apply what we have learned to build a realistic, configuration-driven application. We will build a log file analyzer that reads its configuration from a TOML file, processes log data from a CSV, and outputs results in JSON.

23.6.1 The Configuration File

Listing 23.19: analyzer.toml — configuration file

```
// Suppose analyzer.toml contains:
// [input]
// path = "logs/access.csv"
// delimiter = ","
//
// [filters]
// min_status = 400
// methods = ["GET", "POST"]
//
// [output]
// format = "json"
// path = "reports/errors.json"
// pretty = true
```

23.6.2 The Analyzer

Listing 23.20: A configuration-driven log analyzer

```
fn load_config(path: String) -> Map {
    let text = read_file(path)
    return toml_parse(text)
}

fn analyze_logs(config: Map) {
    // Read input
    let input_path = config["input"]["path"]
    let csv_text = read_file(input_path)
    let rows = csv_parse(csv_text)

    if rows.len() < 2 {
        print("No data to analyze")
        return
    }

    let headers = rows[0]
    let min_status = config["filters"]["min_status"]
    let allowed_methods = config["filters"]["methods"]

    // Find column indices
    flux status_col = -1
    flux method_col = -1
    flux path_col = -1
    flux i = 0
    while i < headers.len() {
        if headers[i] == "status" { status_col = i }
        if headers[i] == "method" { method_col = i }
        if headers[i] == "path" { path_col = i }
        i += 1
    }

    // Filter and collect matching rows
    flux errors = []
    i = 1
    while i < rows.len() {
        let row = rows[i]
        let status = parse_int(row[status_col])
        let method = row[method_col]

        if status >= min_status {
            flux method_match = false
            for allowed in allowed_methods {
                if method == allowed {
                    method_match = true
                    break
```

This program demonstrates the power of combining formats: TOML for human-readable configuration, CSV for tabular data ingestion, and JSON for structured output.

23.6.3 Using the dotenv Library

For programs that need environment-variable configuration (common in twelve-factor apps and containerized deployments), Lattice ships with a `dotenv` library that parses `.env` files:

Listing 23.21: Loading environment variables from a .env file

```
import "lib/dotenv" as dotenv

// Load .env from the current directory
dotenv.load()

// Access environment variables
let db_url = env("DATABASE_URL")
let secret = env("SECRET_KEY")
print("Connecting to: " + db_url)
```

The dotenv library, found in `lib/dotenv.lat`, supports double-quoted values with escape sequences, single-quoted literal values, inline comments, the `export` prefix, multiline values, and even variable expansion (`${VAR}` syntax). It is a good example of a real-world configuration parser built entirely in Lattice.

Layered Configuration

A robust configuration strategy combines multiple sources with clear precedence: environment variables override `.env` files, which override `config.toml` defaults. The dotenv library's `load_opts` function respects this pattern—by default it does *not* override variables that are already set in the environment.

Exercises

1. **JSON Pretty Printer.** Write a function `json_pretty(value: any) -> String` that produces indented, human-readable JSON output. Use recursion to handle nested maps and arrays, adding 2 spaces of indentation per level.

2. **Config Merger.** Write a function that takes two TOML configuration strings, parses both, and deep-merges them—the second configuration's values override the first. Nested maps should be merged recursively rather than replaced wholesale.

3. **CSV Column Filter.** Write a program that reads a CSV file, keeps only the columns specified in a TOML configuration file, and writes the filtered CSV to a new file. The config should look like: `columns = ["name", "email", "department"]`.

4. **Schema Validator.** Write a function that takes a parsed JSON value and a "schema" (a Map describing expected types for each key) and returns an array of validation errors. For example, the schema `\{"name": "String", "age": "Int"\}` should report an error if `name` is missing or `age` is a string.

5. **Format Converter CLI.** Write a program that reads from standard input in one format and writes to standard output in another. Accept the input and output formats as command-line arguments: `lattice convert.lat -from yaml -to json < input.yaml`.

What's Next

We can now read files and parse structured data. The next frontier is communicating over the network. In the next chapter, we will explore Lattice's HTTP client for making web requests, TCP and TLS networking for lower-level communication, and how to build a working HTTP server—combining everything we have learned about strings, data formats, and concurrency.

Chapter 24

Networking and HTTP

Most modern programs need to talk to the network. Whether you are fetching data from a REST API, scraping a web page, or building a microservice, networking is the backbone. Lattice provides networking at two levels: a high-level HTTP client that handles the protocol details for you, and a low-level TCP/TLS layer that gives you direct control over sockets. On top of these primitives, the `http_server` library offers an Express-style framework for building web servers entirely in Lattice.

We will start at the top—making HTTP requests—and then work our way down to raw sockets, TLS, and server construction.

24.1 The http Module

The HTTP functions are available both as top-level built-ins (`http_get`, `http_post`, `http_request`) and as methods on the `http` module object. They handle URL parsing, connection management, TLS negotiation for HTTPS, request formatting, chunked transfer decoding, and response parsing—all behind a single function call.

24.1.1 Simple GET Requests

Listing 24.1: Making an HTTP GET request

```
let response = http_get("https://httpbin.org/get")

print(response["status"])   // 200
print(response["body"])     // JSON response body

// Response headers are a Map with lowercase keys
let headers = response["headers"]
print(headers["content-type"])  // "application/json"
```

`http_get` takes a URL string and returns a `Map` with three keys:

Key	Type	Description
`"status"`	`Int`	HTTP status code (200, 404, 500, etc.)
`"body"`	`String`	Response body as a string
`"headers"`	`Map`	Response headers (keys are lowercase)

The URL must start with `http://` or `https://`. If the scheme is `https`, the HTTP layer automatically performs a TLS handshake with certificate verification before sending the request—you do not need to manage TLS yourself.

Listing 24.2: Fetching and parsing JSON from an API

```
let response = http_get("https://api.example.com/users")

if response["status"] == 200 {
    let users = json_parse(response["body"])
    for user in users {
        print(user["name"] + " (" + user["email"] + ")")
    }
} else {
    print("Request failed with status " + to_string(response["status"]))
}
```

HTTP and the Filesystem Chapter

Notice how naturally HTTP combines with the data format functions from Chapter 23. The response body is always a `String`; use `json_parse`, `yaml_parse`, or the other parsers to turn it into structured data.

24.1.2 POST Requests

Listing 24.3: Sending a POST request with a JSON body

```
let payload = Map::new()
payload["username"] = "alice"
payload["email"] = "alice@example.com"

let options = Map::new()
options["body"] = json_stringify(payload)
options["headers"] = Map::new()
options["headers"]["Content-Type"] = "application/json"

let response = http_post("https://api.example.com/users", options)
print(response["status"])  // 201
print(response["body"])
```

`http_post` takes a URL and an optional options Map. The options Map can include:

- `"body"` — the request body as a String
- `"headers"` — a Map of additional request headers
- `"timeout"` — request timeout in milliseconds (default: 30 000)

24.1.3 The General-Purpose http_request

For methods beyond GET and POST—PUT, PATCH, DELETE, HEAD—use the general-purpose `http_request`:

Listing 24.4: Using http_request for other methods

```
// PUT request to update a resource
let update_body = json_stringify({"name": "Alice Updated"})
let options = Map::new()
options["body"] = update_body
options["headers"] = Map::new()
options["headers"]["Content-Type"] = "application/json"

let response = http_request("PUT", "https://api.example.com/users/1", options)
print(response["status"])  // 200

// DELETE request
let del_response = http_request("DELETE", "https://api.example.com/users/1", {})
print(del_response["status"])  // 204
```

`http_request` takes the HTTP method as a string, the URL, and an optional options Map with the same keys as `http_post`.

Timeouts

All HTTP functions use a default timeout of 30 seconds. For slow APIs or large downloads, pass a `"timeout"` key in the options Map:

```
let options = Map::new()
options["timeout"] = 60000  // 60 seconds
let response = http_get("https://slow-api.example.com/data")
```

A timeout of 0 means "use the default" (30 seconds), not "no timeout."

24.1.4 Error Handling in HTTP

HTTP functions raise a runtime error when something goes wrong at the *network* level: DNS resolution failure, connection refused, TLS certificate validation failure, or timeout. HTTP-level errors (4xx, 5xx status codes) are *not* runtime errors—they are indicated by the `"status"` field in the response. This means you should always check the status code:

Listing 24.5: Robust HTTP error handling

```
fn fetch_user(user_id: Int) -> Map {
    let url = "https://api.example.com/users/" + to_string(user_id)

    let response = try {
        http_get(url)
    } catch error {
        print("Network error: " + to_string(error))
        return nil
    }

    if response["status"] != 200 {
        print("API error: " + to_string(response["status"]))
        return nil
    }

    return json_parse(response["body"])
}
```

HTTPS Certificate Verification

Lattice's TLS layer verifies server certificates against the system's trusted certificate store. If you connect to a server with an expired, self-signed, or otherwise invalid certificate, the request will fail with a TLS error. This is the correct, safe default—do not work around it in production.

24.2 The net Module: TCP Networking

Beneath the HTTP layer, Lattice exposes raw TCP socket operations. These functions work with integer *socket descriptors*—opaque handles that identify a connection. The net layer tracks which descriptors are valid sockets, so passing a random integer will produce a clear error rather than undefined behavior.

24.2.1 Connecting to a Server

Listing 24.6: Connecting to a TCP server

```
let fd = tcp_connect("example.com", 80)

// Send an HTTP request manually
let request = "GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n"
tcp_write(fd, request)

// Read the response
let response = tcp_read(fd)
print(response)

tcp_close(fd)
```

`tcp_connect` performs DNS resolution and establishes a TCP connection. It returns an integer socket descriptor. `tcp_write` sends data over the connection, handling partial writes internally. `tcp_read` reads up to 8 KB of available data and returns it as a string. An empty string indicates end-of-file (the remote side closed the connection).

Listing 24.7: Reading all data from a TCP connection

```
fn read_all(fd: Int) -> String {
    flux result = ""
    loop {
        let chunk = tcp_read(fd)
        if chunk.len() == 0 {
            break  // EOF
        }
        result = result + chunk
    }
    return result
}
```

If you need to read an exact number of bytes, use `tcp_read_bytes`:

Listing 24.8: Reading an exact number of bytes

```
let fd = tcp_connect("example.com", 80)
tcp_write(fd, "GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n")

// Read exactly 1024 bytes (or less if the connection closes)
let data = tcp_read_bytes(fd, 1024)
print("Received " + to_string(data.len()) + " bytes")

tcp_close(fd)
```

24.2.2 Building a TCP Server

To accept incoming connections, use `tcp_listen` and `tcp_accept`:

Listing 24.9: A minimal TCP echo server

```
let server_fd = tcp_listen("0.0.0.0", 9000)
print("Echo server listening on port 9000")

loop {
    let client_fd = tcp_accept(server_fd)
    let peer = tcp_peer_addr(client_fd)
    print("Connection from " + peer)

    // Read what the client sends
    let data = tcp_read(client_fd)

    // Echo it back
    tcp_write(client_fd, data)

    tcp_close(client_fd)
    print("Closed connection from " + peer)
}
```

`tcp_listen` creates a server socket bound to the given address and port, with `SO_REUSEADDR` set so you can restart the server without waiting for the socket to time out. `tcp_accept` blocks until a client connects, then returns a new socket descriptor for that connection. `tcp_peer_addr` returns the remote address as a `"ip:port"` string.

Socket Timeouts

Use `tcp_set_timeout(fd, seconds)` to set read and write timeouts on a socket. This is important for servers—without a timeout, a misbehaving client that never sends data will block your server forever:

```
let client_fd = tcp_accept(server_fd)
tcp_set_timeout(client_fd, 10)  // 10-second timeout
```

24.2.3 TCP Function Reference

Function	Description
`tcp_listen(host, port)`	Bind and listen; returns server socket descriptor
`tcp_accept(server_fd)`	Accept a connection; returns client socket descriptor
`tcp_connect(host, port)`	Connect to a remote server; returns socket descriptor
`tcp_read(fd)`	Read up to 8 KB; returns String (empty on EOF)
`tcp_read_bytes(fd, n)`	Read exactly `n` bytes; returns String
`tcp_write(fd, data)`	Write all data; handles partial writes
`tcp_close(fd)`	Close socket and release tracking
`tcp_peer_addr(fd)`	Get remote `"ip:port"` as String
`tcp_set_timeout(fd, secs)`	Set read/write timeout in seconds

24.3 TLS: Secure Connections

For encrypted communication, Lattice provides TLS functions that mirror the TCP API but add encryption and certificate verification. The TLS layer is built on OpenSSL (or Schannel on Windows) and supports TLS 1.2 and above.

24.3.1 Making a TLS Connection

Listing 24.10: Connecting with TLS

```
// Check if TLS support is available in this build
if !tls_available() {
    print("TLS not available -- build with OpenSSL support")
    return
}

let fd = tls_connect("example.com", 443)

let request = "GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n"
tls_write(fd, request)

flux response = ""
loop {
    let chunk = tls_read(fd)
    if chunk.len() == 0 { break }
    response = response + chunk
}

print(response)
tls_close(fd)
```

`tls_connect` performs a TCP connection, then negotiates a TLS handshake on top of it. It sets the Server Name Indication (SNI) extension and enables hostname verification, so the connection is secure by default. The returned descriptor is the same integer type as TCP descriptors, but you *must* use the `tls_*` functions to read and write—the data must pass through the encryption layer.

Don't Mix TCP and TLS Functions

A descriptor returned by `tls_connect` must be used with `tls_read`, `tls_write`, and `tls_close`. If you call `tcp_read` on a TLS descriptor, you will read encrypted gibberish. If you call `tcp_close` instead of `tls_close`, the TLS session will not be properly shut down.

24.3.2 TLS Function Reference

Function	Description
`tls_connect(host, port)`	TCP connect + TLS handshake + cert verification
`tls_read(fd)`	Read up to 8 KB of decrypted data
`tls_read_bytes(fd, n)`	Read exactly n decrypted bytes
`tls_write(fd, data)`	Encrypt and write all data
`tls_close(fd)`	TLS shutdown + close socket
`tls_available()`	Returns `true` if TLS support was compiled in

When to Use TLS Directly

In practice, you rarely need the TLS functions directly. `http_get` and friends handle TLS automatically for `https://` URLs. The TLS layer is useful when you need to communicate over encrypted sockets with non-HTTP protocols: database connections, message queues, custom binary protocols, or connecting to services like Redis over TLS.

24.4 Building a Simple HTTP Server

Now that we understand both the HTTP client and the TCP layer, let's build something on the server side. Lattice ships with two approaches: you can build a server from scratch using raw TCP (instructive), or use the `http_server` library for a more productive, framework-style experience.

24.4.1 A Server from Scratch

The minimal approach constructs HTTP responses as raw strings:

Listing 24.11: A minimal HTTP server from raw TCP

```
fn build_response(status: Int, status_text: String,
                  content_type: String, body: String) -> String {
    let header = "HTTP/1.1 " + to_string(status) + " " + status_text + "\r\n"
        + "Content-Type: " + content_type + "\r\n"
        + "Content-Length: " + to_string(body.len()) + "\r\n"
        + "Connection: close\r\n"
        + "\r\n"
    return header + body
}

let server = tcp_listen("0.0.0.0", 8080)
print("Listening on http://localhost:8080")

loop {
    let conn = tcp_accept(server)
    let raw = tcp_read(conn)

    if raw.len() > 0 {
        // Parse the request line: "GET /path HTTP/1.1"
        let lines = raw.split("\r\n")
        let parts = lines[0].split(" ")
        let method = parts[0]
        let path = if parts.len() > 1 { parts[1] } else { "/" }

        print(method + " " + path)

        let response = if path == "/" {
            build_response(200, "OK", "text/html",
                "<h1>Hello from Lattice!</h1>")
        } else if path == "/json" {
            build_response(200, "OK", "application/json",
                json_stringify({"message": "Hello!"}))
        } else {
            build_response(404, "Not Found", "text/plain",
                "404 Not Found: " + path)
        }

        tcp_write(conn, response)
    }
    tcp_close(conn)
}
```

This works, but it has obvious limitations: no routing, no middleware, no cookie handling, and no request body parsing. The `http_server` library handles all of that.

24.4.2 The http_server Library

Import the library and create an application:

Listing 24.12: Creating an HTTP server with the http_server library

```
import "lib/http_server" as http

let app = http.new()
```

Registering Routes

Routes are registered by HTTP method:

Listing 24.13: Registering route handlers

```
app.get("/", |req, res| {
    return http.response(200, "Hello, World!")
})

app.get("/about", |req, res| {
    return http.html(200, "<h1>About</h1><p>Built with Lattice.</p>")
})

app.post("/api/items", |req, res| {
    let body = req.get("body")
    print("Received: " + body)
    return http.json(201, {"status": "created"})
})

app.delete("/api/items", |req, res| {
    return http.json(200, {"status": "deleted"})
})
```

Route handlers receive a request Map and a response context Map. They must return a response Map, which you build using the helper functions:

Helper	Description
`http.response(status, body)`	Plain text response
`http.json(status, data)`	JSON response (auto-stringifies the data)
`http.html(status, body)`	HTML response
`http.redirect(url)`	302 redirect (optional status parameter)
`http.stream(status, chunks)`	Chunked transfer encoding response

The Request Map

The request Map passed to handlers contains:

Key	Type	Description
`"method"`	`String`	HTTP method ("GET", "POST", etc.)
`"path"`	`String`	Request path without query string
`"query"`	`Map`	Parsed query parameters
`"headers"`	`Map`	Request headers
`"body"`	`String`	Request body
`"cookies"`	`Map`	Parsed cookies
`"peer"`	`String`	Remote address (`"ip:port"`)

Listing 24.14: Working with query parameters and cookies

```
app.get("/search", |req, res| {
    let query = req.get("query")
    let term = if query.has("q") { query.get("q") } else { "" }
    return http.json(200, {"results": [], "query": term})
})

app.get("/profile", |req, res| {
    let cookies = req.get("cookies")
    let session = if cookies.has("session") {
        cookies.get("session")
    } else {
        "anonymous"
    }
    let response = http.json(200, {"user": session})
    return http.set_cookie(response, "visited", "true", http.cookie_opts())
})
```

Middleware

Middleware functions run before route handlers. Each middleware receives the request, response context, and a `next` function to call the rest of the pipeline:

Listing 24.15: Adding middleware to a server

```
import "lib/http_server" as http

let app = http.new()

// Built-in logging middleware
app.use(http.logger())

// Built-in CORS middleware
app.use(http.cors())

// Built-in security headers
app.use(http.security_headers())

// Built-in JSON body parser
app.use(http.body_parser())

// Custom middleware: authentication
app.use(|req, res, next| {
    let headers = req.get("headers")
    if headers.has("Authorization") {
        req.set("authenticated", true)
    } else {
        req.set("authenticated", false)
    }
    return next(req, res)
})

app.get("/admin", |req, res| {
    if !req.get("authenticated") {
        return http.json(401, {"error": "Unauthorized"})
    }
    return http.json(200, {"message": "Welcome, admin"})
})

app.listen(8080)
```

The middleware pipeline executes in registration order. The `logger` middleware prints timing information for each request. The `cors` middleware handles preflight OPTIONS requests and sets the appropriate `Access-Control-*` headers. The `security_headers` middleware adds `X-Content-Type-Options`, `X-Frame-Options`, and similar defensive headers.

Custom CORS Configuration

The `cors()` middleware accepts an options Map:

```
let cors_config = http.cors_opts()
cors_config["origin"] = "https://mysite.com"
cors_config["credentials"] = true
app.use(http.cors(cors_config))
```

Starting the Server

Listing 24.16: Starting the server

```
app.listen(8080)
// "Lattice HTTP server listening on http://localhost:8080"
```

The `listen` call blocks forever, accepting connections in a loop. Each connection is handled synchronously. For concurrent handling, you could combine this with structured concurrency (Chapter 19), spawning a task for each accepted connection.

24.4.3 A Complete REST API

Let's put it all together with a complete in-memory REST API:

Listing 24.17: A complete REST API server

```
import "lib/http_server" as http

let app = http.new()
app.use(http.logger())
app.use(http.cors())
app.use(http.body_parser())

// In-memory data store
flux items = []
flux next_id = 1

// List all items
app.get("/api/items", |req, res| {
    return http.json(200, items)
})

// Create an item
app.post("/api/items", |req, res| {
    let body = req.get("body")
    let data = json_parse(body)
    data["id"] = next_id
    next_id += 1
    items.push(data)
    return http.json(201, data)
})

// Health check
app.get("/health", |req, res| {
    return http.json(200, {"status": "ok", "items": items.len()})
})

print("Starting API server...")
app.listen(3000)
```

24.5 URL Encoding and Decoding

URLs can only contain a limited set of ASCII characters. When you need to include special characters—spaces, ampersands, Unicode text—you must percent-encode them. Lattice provides `url_encode` and `url_decode` for this purpose.

Listing 24.18: URL encoding and decoding

```
let query = "name=Alice Bob&city=New York"
let encoded = url_encode(query)
print(encoded)  // "name%3DAlice%20Bob%26city%3DNew%20York"

let decoded = url_decode(encoded)
print(decoded)  // "name=Alice Bob&city=New York"
```

`url_encode` percent-encodes every character that is not an unreserved character (A–Z, a–z, 0–9, hyphen, underscore, period, tilde). `url_decode` reverses the process, converting `%XX` sequences back to the original bytes and converting `+` to spaces.

Listing 24.19: Building a URL with query parameters

```
fn build_url(base: String, params: Map) -> String {
    let keys = params.keys()
    if keys.len() == 0 {
        return base
    }

    flux parts = []
    for key in keys {
        let encoded_key = url_encode(key)
        let encoded_val = url_encode(params[key])
        parts.push(encoded_key + "=" + encoded_val)
    }

    return base + "?" + parts.join("&")
}

let params = Map::new()
params["q"] = "lattice programming"
params["page"] = "1"
params["lang"] = "en"

let url = build_url("https://search.example.com/api", params)
print(url)
// https://search.example.com/api?q=lattice%20programming&page=1&lang=en
```

Module vs. Top-Level Access

`url_encode` and `url_decode` are available both as top-level built-in functions and as methods on the `crypto` module (since they are conceptually about encoding/decoding). The top-level versions are convenient for scripts; the module versions are better for explicit imports in larger programs.

Exercises

1. **HTTP Health Checker.** Write a program that takes a list of URLs from a configuration file, makes an HTTP GET request to each one, and reports whether the service is up (2xx status), degraded (5xx), or unreachable (network error). Use `try`/`catch` to handle connection failures gracefully.

2. **Web Scraper.** Write a function that fetches a web page with `http_get`, extracts all URLs from `href="..."` attributes using string operations (or the regex module from Chapter 25), and returns them as an array. Bonus: follow the links one level deep and report the status of each.

3. **TCP Chat Server.** Extend the echo server from Section 24.2.2 into a simple chat server. When a client connects, ask for their name, then broadcast each message to all connected clients. You will need to use structured concurrency (Chapter 19) to handle multiple clients simultaneously.

4. **REST API with Persistence.** Extend the REST API from Section 24.4.3 to persist its data to a JSON file. Load the file on startup and save after each modification. Use the safe-write pattern from Section 22.4.1 in the filesystem chapter.

5. **HTTP File Server.** Build a server that serves static files from a directory. Map the URL path to a file path, read the file, determine the content type from the extension, and send it back. Handle directory traversal attacks by checking that the resolved path stays within the root directory (use `realpath`).

What's Next

We have now covered the major I/O systems in Lattice: filesystems, data formats, and networking. In the final chapter of this Part, we will survey the remaining standard library modules—time and date handling, cryptographic functions, UUIDs, operating system interaction, mathematics, and regular expressions—giving you the complete toolkit for building real-world applications.

Chapter 25

Time, Crypto, OS, and Everything Else

This chapter is a grand tour of the remaining standard library modules—the functions that do not fit neatly into files, formats, or networking, but that you will reach for constantly in real programs. We will cover timestamps and date arithmetic, cryptographic hashing and encoding, UUID generation, operating-system interaction, mathematics, and regular expressions. Each section is self-contained, so feel free to jump to whichever module you need right now and come back to the others later.

25.1 The time Module

Time in Lattice is represented as a single integer: the number of milliseconds since the Unix epoch (January 1, 1970, 00:00:00 UTC). This is the same convention used by JavaScript's `Date.now()`, making it straightforward to exchange timestamps with web APIs.

25.1.1 Getting the Current Time

Listing 25.1: Getting the current timestamp

```
let now = time()
print(now)  // e.g., 1703456789012

// Measure execution time
let start = time()
// ... some computation ...
let elapsed = time() - start
print("Took " + to_string(elapsed) + " ms")
```

`time()` returns the current wall-clock time as an `Int` in milliseconds. Under the hood, it calls `clock_gettime` with `CLOCK_REALTIME` (see `src/time_ops.c`).

25.1.2 Sleeping

Listing 25.2: Pausing execution

```
print("Starting...")
sleep(1000)  // Sleep for 1 second
print("Done!")

// Short delay for polling
sleep(100)  // 100 milliseconds
```

`sleep` takes a duration in milliseconds. It uses `nanosleep` for precise timing and handles `EINTR` gracefully.

sleep Is Not Available in WASM

In the browser (Emscripten) build, `sleep` raises a runtime error because it would block the single-threaded event loop. If you need time-delayed behavior in WASM, use callbacks or an async pattern instead.

25.1.3 Formatting Timestamps

Listing 25.3: Formatting a timestamp as a human-readable string

```
let now = time()

let readable = time_format(now, "%Y-%m-%d %H:%M:%S")
print(readable)  // "2024-12-25 14:30:45"

let date_only = time_format(now, "%B %d, %Y")
print(date_only)  // "December 25, 2024"

let time_only = time_format(now, "%I:%M %p")
print(time_only)  // "02:30 PM"
```

`time_format` takes a timestamp (in epoch milliseconds) and a format string using the standard `strftime` specifiers:

Specifier	Meaning	Specifier	Meaning
%Y	4-digit year	%H	Hour (00–23)
%m	Month (01–12)	%I	Hour (01–12)
%d	Day (01–31)	%M	Minute (00–59)
%B	Full month name	%S	Second (00–59)
%b	Abbreviated month	%p	AM/PM
%A	Full weekday name	%Z	Timezone abbreviation
%a	Abbreviated weekday	%%	Literal %

25.1.4 Parsing Timestamps

Listing 25.4: Parsing a date string into a timestamp

```
let timestamp = time_parse("2024-12-25 14:30:00", "%Y-%m-%d %H:%M:%S")
print(timestamp)  // epoch milliseconds

let formatted = time_format(timestamp, "%B %d, %Y")
print(formatted)  // "December 25, 2024"
```

`time_parse` is the inverse of `time_format`. It takes a date string and a format pattern, and returns the corresponding epoch timestamp in milliseconds. The format specifiers are the same as for `time_format`.

Local Time

Both `time_format` and `time_parse` work in the system's local timezone by default. The underlying C functions (`localtime_r` and `mktime`) handle timezone conversions and daylight saving time adjustments automatically.

25.1.5 Extracting Date Components

For quick access to individual components without constructing a format string, Lattice provides dedicated extraction functions:

Listing 25.5: Extracting date and time components

```
let now = time()

print(time_year(now))     // 2024
print(time_month(now))    // 12 (1-based)
print(time_day(now))      // 25
print(time_hour(now))     // 14
print(time_minute(now))   // 30
print(time_second(now))   // 45
print(time_weekday(now))  // 3 (0=Sunday, 1=Monday, ..., 6=Saturday)
```

25.1.6 Date Arithmetic

Since timestamps are just integers in milliseconds, date arithmetic is straightforward:

Listing 25.6: Date arithmetic with timestamps

```
let now = time()

// Add 7 days
let next_week = time_add(now, 7 * 24 * 60 * 60 * 1000)
print("Next week: " + time_format(next_week, "%Y-%m-%d"))

// Subtract 30 days
let last_month = time_add(now, -30 * 24 * 60 * 60 * 1000)
print("30 days ago: " + time_format(last_month, "%Y-%m-%d"))

// Useful constants (in milliseconds)
let SECOND = 1000
let MINUTE = 60 * SECOND
let HOUR = 60 * MINUTE
let DAY = 24 * HOUR

let deadline = time_add(now, 3 * HOUR + 30 * MINUTE)
print("Deadline: " + time_format(deadline, "%H:%M:%S"))
```

`time_add` simply adds a delta in milliseconds to a timestamp. You can also use plain addition (`now + delta`), but `time_add` makes the intent clearer.

25.1.7 The datetime Functions

For more structured date/time work—especially when dealing with ISO 8601 strings and UTC conversions—Lattice provides a family of `datetime_*` functions. These work with epoch *seconds* (not milliseconds) and return Maps with structured fields.

Listing 25.7: Working with ISO 8601 dates

```
// Get current datetime as a Map
let now = datetime_now()
print(now)  // Map with year, month, day, hour, minute, second, etc.

// Parse an ISO 8601 string
let dt = datetime_from_iso("2024-12-25T14:30:00Z")
print(dt)

// Convert back to ISO 8601
let iso_str = datetime_to_iso(dt)
print(iso_str)  // "2024-12-25T14:30:00Z"

// Format with a custom pattern
let formatted = datetime_format(dt, "%B %d, %Y at %H:%M")
print(formatted)  // "December 25, 2024 at 14:30"
```

Listing 25.8: Duration arithmetic with datetime

```
let start = datetime_from_iso("2024-01-15T09:00:00Z")

// Add a duration (in seconds)
let meeting_end = datetime_add_duration(start, 90 * 60)  // 90 minutes

// Compute the difference between two datetimes (in seconds)
let end = datetime_from_iso("2024-01-15T10:30:00Z")
let diff = datetime_sub(end, start)
print("Duration: " + to_string(diff / 60) + " minutes")  // 90 minutes
```

The `datetime_to_utc` and `datetime_to_local` functions convert between UTC and local time. `timezone_offset()` returns the local timezone offset from UTC in seconds.

Which Time API to Use?

For simple timing and logging, the `time()` / `time_format()` functions are sufficient. For applications that exchange dates with APIs, store them in databases, or perform calendar arithmetic across timezones, use the `datetime_*` family with ISO 8601 strings.

25.2 The crypto Module

Lattice provides cryptographic primitives for hashing, message authentication, encoding, and random number generation. When built with OpenSSL support (`LATTICE_HAS_TLS`), these functions use OpenSSL's optimized implementations. Without OpenSSL, Lattice falls back to pure-C implementations that produce identical results—just more slowly.

25.2.1 Hashing: sha256, sha512, md5

Listing 25.9: Computing cryptographic hashes

```
let message = "Hello, Lattice!"

let hash = sha256(message)
print(hash)  // 64-character hex string
// e.g., "a1b2c3d4e5f6..."

let long_hash = sha512(message)
print(long_hash)  // 128-character hex string

let legacy_hash = md5(message)
print(legacy_hash)  // 32-character hex string
```

All hash functions take a `String` and return a lowercase hexadecimal string. `sha256` produces a 256-bit (64 hex character) digest. `sha512` produces a 512-bit (128 hex character) digest. `md5` produces a 128-bit (32 hex character) digest.

MD5 Is Not Secure

MD5 is cryptographically broken and should never be used for security purposes (password hashing, digital signatures, integrity verification of untrusted data). It is included for backward compatibility with legacy systems. Use `sha256` or `sha512` for new applications.

Listing 25.10: File integrity checking with sha256

```
fn file_checksum(path: String) -> String {
    let contents = read_file(path)
    return sha256(contents)
}

let expected = "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
let actual = file_checksum("important_data.txt")

if actual == expected {
    print("File integrity verified")
} else {
    print("WARNING: File has been modified!")
}
```

25.2.2 HMAC: hmac_sha256

HMAC (Hash-based Message Authentication Code) combines a secret key with a hash function to verify both the integrity and authenticity of a message:

Listing 25.11: Computing an HMAC

```
let secret_key = "my-secret-key"
let message = "user=alice&amount=100"

let mac = hmac_sha256(secret_key, message)
print(mac)  // 64-character hex string

// Verify by recomputing
let expected_mac = hmac_sha256(secret_key, message)
if mac == expected_mac {
    print("Message is authentic")
}
```

`hmac_sha256` takes a key and a message, both as strings, and returns a hex-encoded MAC. This is the standard way to sign API requests and verify webhook payloads.

25.2.3 Base64 Encoding and Decoding

Base64 converts arbitrary binary data into a text-safe ASCII representation:

Listing 25.12: Base64 encoding and decoding

```
let original = "Hello, World!"

let encoded = base64_encode(original)
print(encoded)  // "SGVsbG8sIFdvcmxkIQ=="

let decoded = base64_decode(encoded)
print(decoded)  // "Hello, World!"
```

Base64 is commonly used for embedding binary data in JSON, passing data in URL parameters, and encoding credentials for HTTP Basic authentication:

Listing 25.13: HTTP Basic authentication with Base64

```
let username = "admin"
let password = "s3cret"
let credentials = base64_encode(username + ":" + password)

let options = Map::new()
options["headers"] = Map::new()
options["headers"]["Authorization"] = "Basic " + credentials

let response = http_get("https://api.example.com/admin")
```

25.2.4 Random Bytes

For cryptographically secure random data, use `random_bytes`:

Listing 25.14: Generating random bytes

```
let token_bytes = random_bytes(32)
print(token_bytes.to_hex())  // 64-character random hex string

// Generate a random hex token
fn generate_token(length: Int) -> String {
    let bytes = random_bytes(length)
    return bytes.to_hex()
}

let session_token = generate_token(16)
print("Session: " + session_token)
```

`random_bytes` returns a `Buffer` of the specified length, filled with cryptographically secure random bytes. On macOS and BSD, it uses `arc4random_buf`. On Linux, it reads from `/dev/urandom`. On Windows, it uses `RtlGenRandom`. With OpenSSL available, it uses `RAND_bytes`. All of these are suitable for generating session tokens, encryption keys, and nonces.

Cryptographic vs. Pseudorandom

`random_bytes` produces cryptographically secure random data—it is unpredictable and suitable for security-sensitive applications. The `random()` function from the math module (see Section 25.5.5) uses `rand()`, which is *not* cryptographically secure and should only be used for non-security purposes like games and simulations.

25.3 uuid()

Universally Unique Identifiers (UUIDs) are 128-bit identifiers typically displayed as 32 hexadecimal digits in a 8-4-4-4-12 format. Lattice provides a built-in function for generating Version 4 (random) UUIDs:

Listing 25.15: Generating UUIDs

```
let id = uuid()
print(id)  // "550e8400-e29b-41d4-a716-446655440000" (example)

// Generate multiple unique identifiers
flux ids = []
flux i = 0
while i < 5 {
    ids.push(uuid())
    i += 1
}
for id in ids {
    print(id)
}
```

Each call to `uuid()` returns a fresh, unique string. The underlying implementation uses `random_bytes` to generate 16 random bytes, then sets the version bits (0100 for V4) and variant bits (10xx for RFC 4122) before formatting.

Listing 25.16: UUIDs for record identifiers

```
fn create_user(name: String, email: String) -> Map {
    let user = Map::new()
    user["id"] = uuid()
    user["name"] = name
    user["email"] = email
    user["created_at"] = time_format(time(), "%Y-%m-%dT%H:%M:%SZ")
    return user
}

let alice = create_user("Alice", "alice@example.com")
print(json_stringify(alice))
```

25.4 The os Module

The operating system functions let your program interact with its environment: reading and setting environment variables, inspecting command-line arguments, executing external commands, and querying system information.

25.4.1 Environment Variables: env, env_set, env_keys

Listing 25.17: Working with environment variables

```
// Read an environment variable
let home = env("HOME")
print("Home directory: " + home)

let path = env("PATH")
print("PATH: " + path)

// Check for a variable (returns nil if not set)
let debug = env("DEBUG")
if debug != nil {
    print("Debug mode is on: " + debug)
} else {
    print("Debug mode is off")
}

// Set an environment variable
env_set("APP_MODE", "production")
print(env("APP_MODE"))  // "production"

// List all environment variable names
let keys = env_keys()
print("Environment has " + to_string(keys.len()) + " variables")
```

`env(name)` returns the value of the named environment variable as a `String`, or `nil` if it is not set. `env_set` sets a variable for the current process (it does not modify the parent shell's environment). `env_keys` returns an array of all environment variable names.

25.4.2 Command-Line Arguments: args()

Listing 25.18: Accessing command-line arguments

```
// Run: lattice script.lat --verbose output.txt

let arguments = args()
print(arguments)  // ["script.lat", "--verbose", "output.txt"]

// Simple argument parsing
flux verbose = false
flux output_path = "default.txt"

flux i = 0
while i < arguments.len() {
    if arguments[i] == "--verbose" {
        verbose = true
    } else if !arguments[i].starts_with("--") {
        output_path = arguments[i]
    }
    i += 1
}

if verbose {
    print("Writing to: " + output_path)
}
```

`args()` returns the command-line arguments as an array of strings. The first element is typically the script name.

25.4.3 System Information: platform, cwd, hostname, pid

Listing 25.19: Querying system information

```
print("Platform: " + platform())     // "macos", "linux", "windows", or "wasm"
print("Working dir: " + cwd())       // "/home/user/project"
print("Hostname: " + hostname())     // "workstation"
print("Process ID: " + to_string(pid()))  // 12345
```

`platform()` returns a short string identifying the operating system. This is useful for writing cross-platform scripts that need to adjust their behavior:

Listing 25.20: Platform-specific behavior

```
let separator = if platform() == "windows" { "\\" } else { "/" }
let config_dir = if platform() == "macos" {
    env("HOME") + "/Library/Application Support/myapp"
} else if platform() == "linux" {
    env("HOME") + "/.config/myapp"
} else {
    env("APPDATA") + "\\myapp"
}

if !file_exists(config_dir) {
    mkdir(config_dir)
}
```

25.4.4 Executing External Commands: exec and shell

Listing 25.21: Running external commands with exec

```
// exec runs a command and returns its stdout as a string
let output = exec("echo 'Hello from the shell'")
print(output)  // "Hello from the shell\n"

// Capture the output of a system command
let git_status = exec("git status --short")
print(git_status)
```

`exec` runs a command via popen, captures its standard output, and returns it as a string. If the command exits with a non-zero status, `exec` raises a runtime error.

For more control—including access to both stdout, stderr, and the exit code—use `shell`:

Listing 25.22: Running commands with shell for full control

```
let result = shell("ls -la /nonexistent")

print("Exit code: " + to_string(result["exit_code"]))
print("Stdout: " + result["stdout"])
print("Stderr: " + result["stderr"])

// Check if a command succeeded
if result["exit_code"] == 0 {
    print("Success!")
} else {
    print("Failed: " + result["stderr"])
}
```

`shell` returns a Map with three keys: `"exit_code"` (an `Int`), `"stdout"` (a `String`), and `"stderr"` (a `String`). Unlike `exec`, it does *not* raise an error on non-zero exit codes, giving you full control over error handling.

Shell Injection

Both `exec` and `shell` pass the command string directly to the system shell (`/bin/sh` on Unix, `cmd.exe` on Windows). **Never** construct command strings from untrusted user input without sanitization—this is a classic shell injection vulnerability:

```
// DANGEROUS: user_input could contain "; rm -rf /"
let result = exec("echo " + user_input)

// SAFER: validate or escape the input first
```

25.5 The math Module

Lattice's math functions are registered as top-level built-ins. They accept both `Int` and `Float` arguments (promoting to `Float` when needed) and cover the standard mathematical operations.

25.5.1 Rounding and Absolute Value

Listing 25.23: Rounding functions

```
print(abs(-42))       // 42
print(abs(-3.14))     // 3.14

print(floor(3.7))     // 3
print(floor(-2.3))    // -3

print(ceil(3.2))      // 4
print(ceil(-2.7))     // -2

print(round(3.5))     // 4
print(round(2.4))     // 2
```

`floor`, `ceil`, and `round` all return `Int` values (the C `floor`/`ceil`/`round` cast to `int64_t`). When given an `Int`, they return it unchanged.

25.5.2 Powers, Roots, and Logarithms

Listing 25.24: Powers and roots

```
print(pow(2, 10))     // 1024 (Int)
print(pow(2, -1))     // 0.5 (Float, because negative exponent)
print(sqrt(144))      // 12.0

print(log(math_e()))   // 1.0 (natural log)
print(log2(1024))      // 10.0
print(log10(1000))     // 3.0

print(exp(1))          // 2.718281828... (e^1)
```

`pow` is smart about types: when both arguments are non-negative integers, it returns an `Int` (with overflow detection that falls back to `Float`). `sqrt` always returns a `Float` and raises a domain error for negative inputs.

25.5.3 Min, Max, Clamp, and Lerp

Listing 25.25: Clamping and interpolation

```
print(min(3, 7))     // 3
print(max(3, 7))     // 7

// Clamp a value to a range
let health = clamp(120, 0, 100)
print(health)  // 100

let brightness = clamp(-10, 0, 255)
print(brightness)  // 0

// Linear interpolation
let midpoint = lerp(0.0, 100.0, 0.5)
print(midpoint)  // 50.0

let quarter = lerp(10.0, 20.0, 0.25)
print(quarter)  // 12.5
```

`clamp(value, low, high)` constrains a value to the range [*low*, *high*]. `lerp(a, b, t)` computes the linear interpolation $a + (b - a) \times t$. Both are essential for game development, animation, and signal processing.

25.5.4 Trigonometry

Listing 25.26: Trigonometric functions

```
let pi = math_pi()

print(sin(0))           // 0.0
print(sin(pi / 2))      // 1.0
print(cos(0))           // 1.0
print(cos(pi))          // -1.0
print(tan(pi / 4))      // ~1.0

// Inverse trig
print(asin(1.0))        // ~1.5707... (pi/2)
print(acos(0.0))        // ~1.5707... (pi/2)
print(atan(1.0))        // ~0.7853... (pi/4)

// atan2 for angle from coordinates
let angle = atan2(1.0, 1.0)
print(angle)  // ~0.7853... (pi/4, i.e., 45 degrees)
```

All trigonometric functions work in radians. `math_pi()` returns π and `math_e()` returns Euler's number e, both as `Float` constants.

Hyperbolic functions are also available: `sinh`, `cosh`, and `tanh`.

25.5.5 Random Numbers

Listing 25.27: Generating random numbers

```
// Random float in [0, 1)
let r = random()
print(r)  // e.g., 0.7234...

// Random integer in a range [low, high] (inclusive)
let dice = random_int(1, 6)
print("You rolled: " + to_string(dice))

// Shuffle an array using random
fn shuffle(arr: Array) -> Array {
    flux result = arr
    flux i = result.len() - 1
    while i > 0 {
        let j = random_int(0, i)
        let temp = result[i]
        result[i] = result[j]
        result[j] = temp
        i -= 1
    }
    return result
}

let deck = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let shuffled = shuffle(deck)
print(shuffled)
```

`random()` returns a `Float` in the range $[0, 1)$. `random_int(low, high)` returns an `Int` in the inclusive range $[\text{low}, \text{high}]$.

25.5.6 Other Math Functions

Listing 25.28: Additional math functions

```
// Sign function
print(sign(-42))   // -1
print(sign(0))     // 0
print(sign(7.5))   // 1.0

// Greatest common divisor and least common multiple
print(gcd(12, 8))   // 4
print(lcm(4, 6))    // 12

// Special value detection
print(is_nan(0.0 / 0.0))   // true
print(is_inf(1.0 / 0.0))   // true
print(is_nan(42.0))         // false
```

25.6 The regex Module

Regular expressions let you search, match, and transform text using patterns. Lattice uses POSIX Extended Regular Expressions (ERE) on Unix systems.

25.6.1 Matching: regex_match

Listing 25.29: Testing if a string matches a pattern

```
let email = "alice@example.com"
let pattern = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$"

if regex_match(pattern, email) {
    print("Valid email format")
} else {
    print("Invalid email format")
}
```

`regex_match` returns `true` if the pattern matches anywhere in the string. Use `^` and `$` anchors to require a full-string match.

Listing 25.30: Case-insensitive matching with flags

```
// The third argument is a flags string
let is_greeting = regex_match("^hello", "Hello World", "i")
print(is_greeting)  // true
```

The optional flags string supports:

Flag	Meaning
`i`	Case-insensitive matching (`REG_ICASE`)
`m`	Newline-sensitive matching (`REG_NEWLINE`)

25.6.2 Finding All Matches: regex_find_all

Listing 25.31: Finding all pattern matches in a string

```
let text = "Call 555-1234 or 555-5678 for info"
let phone_pattern = "[0-9]{3}-[0-9]{4}"

let numbers = regex_find_all(phone_pattern, text)
for number in numbers {
    print("Found: " + number)
}
// Found: 555-1234
// Found: 555-5678
```

`regex_find_all` returns an array of all non-overlapping matches. If no matches are found, it returns an empty array.

Listing 25.32: Extracting data from structured text

```
let log = """
[2024-01-15] ERROR: Connection timeout
[2024-01-15] INFO: Server started
[2024-01-16] ERROR: Disk space low
[2024-01-16] WARN: Memory usage high
"""

let errors = regex_find_all("\\[.*\\] ERROR: .*", log)
print("Found " + to_string(errors.len()) + " errors:")
for error in errors {
    print("  " + error)
}
// Found 2 errors:
//   [2024-01-15] ERROR: Connection timeout
//   [2024-01-16] ERROR: Disk space low
```

25.6.3 Replacing: regex_replace

Listing 25.33: Replacing patterns in text

```
let text = "The quick brown fox jumps over the lazy dog"

// Replace all vowels
let result = regex_replace("[aeiou]", text, "*")
print(result)  // "Th* q**ck br*wn f*x j*mps *v*r th* l*zy d*g"

// Sanitize user input: remove non-alphanumeric characters
let input = "Hello! <script>alert('xss')</script>"
let clean = regex_replace("[^a-zA-Z0-9 ]", input, "")
print(clean)  // "Hello scriptalertxssscript"
```

`regex_replace` replaces *all* non-overlapping matches of the pattern with the replacement string. It returns a new string; the original is unmodified.

Listing 25.34: Normalizing whitespace

```
let messy = "Too   many     spaces    here"
let clean = regex_replace("  +", messy, " ")
print(clean)  // "Too many spaces here"
```

Regex on Windows

The regex functions use POSIX `regex.h`, which is not available in MinGW on Windows. On Windows builds, the regex functions return stub errors. A future version may bundle PCRE2 for cross-platform support. See `src/regex_ops.c` for details.

25.6.4 Practical Example: Log Parser

Let's combine regex with file I/O and data formats to build a log file parser:

Listing 25.35: A log parser using regex

```
fn parse_log(path: String) -> Array {
    let content = read_file(path)
    let lines = content.split("\n")

    flux entries = []
    let pattern = "^\\[([0-9-]+)\\] ([A-Z]+): (.*)$"

    for line in lines {
        if line.len() == 0 { continue }

        let matches = regex_find_all(
            "[0-9]{4}-[0-9]{2}-[0-9]{2}", line)
        let levels = regex_find_all("[A-Z]{3,}", line)

        if matches.len() > 0 && levels.len() > 0 {
            let entry = Map::new()
            entry["date"] = matches[0]
            entry["level"] = levels[0]
            entry["message"] = line
            entries.push(entry)
        }
    }

    return entries
}

let entries = parse_log("application.log")
let json_output = json_stringify(entries)
write_file("parsed_log.json", json_output)
print("Parsed " + to_string(entries.len()) + " log entries")
```

Exercises

1. **Stopwatch.** Write a stopwatch utility with functions `start()`, `lap()`, and `stop()` that records split times. Print each lap time and the total elapsed time, formatted as `MM:SS.mmm`.

2. **Password Hasher.** Write a function that takes a password and a salt (generated with `random_bytes`), computes `hmac_sha256(salt, password)`, and returns a string in the format `salt$hash` (where the salt is hex-encoded). Write a companion `verify` function that checks a password against a stored hash.

3. **Cron Expression Evaluator.** Write a function that takes a simple cron-like expression (e.g., `"*/5 * * * *"` for "every 5 minutes") and the current time, and returns the next time the expression will fire. Use the `time_*` extraction functions and date arithmetic.

4. **System Information Report.** Write a program that collects system information—platform, hostname, PID, current directory, all environment variables, and the current time—and outputs it as a formatted YAML document.

5. **Text Statistics.** Write a program that reads a text file and uses `regex_find_all` to count: (a) the number of words, (b) the number of sentences (ending with `.` `!` or `?`), (c) the number of email addresses, and (d) the number of URLs. Output the results as a JSON object.

6. **Monte Carlo Pi Estimator.** Use `random()` to estimate π by the Monte Carlo method: generate random (x, y) points in the unit square, count how many fall inside the unit circle ($x^2 + y^2 \leq 1$), and compute $\pi \approx 4 \times \text{inside}/\text{total}$. How many points do you need for 4 decimal places of accuracy?

What's Next

With this chapter, we have completed our tour of the standard library. You now have the tools to work with files, parse and generate data in multiple formats, communicate over the network, manipulate dates, hash and encode data, generate unique identifiers, interact with the operating system, do mathematics, and match text with regular expressions. In Chapter 26, we will shift gears and explore Lattice's iterator system—a powerful abstraction for processing sequences lazily and composing transformations with a functional style.

Part VII

Iterators and Functional Style

Chapter 26

Lazy Iterators

Imagine you have a log file with ten million lines, and you need the first twenty that contain the word `"ERROR"`. One approach: read all ten million lines into an array, filter it, then slice off the first twenty. That works, but it loads the entire file into memory and does far more work than necessary.

There is a better way. *Iterators* let you process data one element at a time, on demand, without ever building the full collection in memory. In Lattice, iterators are *lazy*—they produce values only when asked. This chapter shows you how to create iterators, chain transformations, consume results, and build your own custom iterators from scratch.

26.1 The Iterator Protocol

At its heart, an iterator is anything that can produce a sequence of values, one at a time, until it is exhausted. Lattice's built-in `iter()` function converts a collection into an `Iterator` value, and the `.next()` method advances it by one step.

Listing 26.1: The basic iterator protocol

```
let colors = ["crimson", "teal", "gold"]
let it = iter(colors)

print(it.next())  // crimson
print(it.next())  // teal
print(it.next())  // gold
print(it.next())  // nil
```

Each call to `.next()` returns the next value. When the iterator is exhausted, it returns `nil`.

Iterator

An *iterator* is a value of type `Iterator` that encapsulates:

1. Some internal **state** (a position, a counter, a reference to a source).
2. A **next function** that, given the current state, either produces the next value or signals that the sequence is done.

Once an iterator is exhausted, every subsequent call to `.next()` returns `nil`. Iterators are single-pass: you cannot rewind them.

26.1.1 What Can You Iterate Over?

The `iter()` function accepts arrays, maps, sets, strings, ranges, and even existing iterators (which it returns unchanged):

Listing 26.2: Creating iterators from different collections

```
// Array: yields elements in order
let arr_it = iter([10, 20, 30])

// String: yields individual characters
let str_it = iter("hello")
print(str_it.next())  // h
print(str_it.next())  // e

// Range: yields integers lazily
let range_it = iter(0..5)
print(range_it.next())  // 0
print(range_it.next())  // 1

// Map: yields keys
let scores = Map::new()
scores.set("alice", 95)
scores.set("bob", 87)
let map_it = iter(scores)
print(map_it.next())  // alice (or bob -- map order is not guaranteed)
```

Iterators Are Single-Pass

Once you call `.next()` and consume a value, it is gone. If you need to iterate over the same data twice, create a new iterator with `iter()` again—or `.collect()` the results into an array first.

26.1.2 Using Iterators in For Loops

You rarely call `.next()` by hand. Lattice's `for` loop consumes any iterable automatically:

Listing 26.3: Iterators and for loops

```
let temperatures = [72.1, 68.4, 75.0, 71.3]

// The for loop calls iter() and .next() behind the scenes
for temp in temperatures {
    print("Reading: " + to_string(temp))
}
```

When you write `for x in collection`, Lattice creates an iterator from the collection and repeatedly calls `.next()` until it returns `nil`. This is the same protocol you just learned—the `for` loop is syntactic sugar for the manual `.next()` loop.

26.2 Sources: Where Iterators Come From

The `iter()` function wraps existing collections, but Lattice also provides dedicated *source* constructors that create iterators from scratch.

26.2.1 iter() — From Collections

As we saw above, `iter()` is the universal entry point. It accepts arrays, maps, sets, strings, ranges, and iterators. Under the hood, `iter()` dispatches to specialized constructors like `iter_from_array`, `iter_from_map`, and `iter_from_range` in the C runtime (see `src/iterator.c`).

26.2.2 range_iter() — Lazy Integer Sequences

While `iter(0..1000)` converts a range to an iterator, the dedicated `range_iter()` function gives you more control—including a custom step:

Listing 26.4: Creating range iterators

```
// Count from 0 to 9
let digits = range_iter(0, 10)
print(digits.collect())  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

// Even numbers from 0 to 20
let evens = range_iter(0, 20, 2)
print(evens.collect())  // [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

// Countdown
let countdown = range_iter(5, 0, -1)
print(countdown.collect())  // [5, 4, 3, 2, 1]
```

The third argument is the step. If omitted, it defaults to `1` for ascending ranges or `-1` for descending ranges. No array is allocated—the iterator computes each value on the fly.

When to Use range_iter() vs. a Range Literal

If you need a custom step or you are building a pipeline of iterator transformations, use `range_iter()`. For straightforward `for` loops, the range literal (`0..10`) is more concise and equally lazy.

26.2.3 repeat_iter() — The Same Value, Over and Over

Need a sequence of identical values? `repeat_iter()` yields the same value a specified number of times—or infinitely if you omit the count:

Listing 26.5: Repeating a value

```
// Repeat a value 4 times
let zeros = repeat_iter(0, 4)
print(zeros.collect())  // [0, 0, 0, 0]

// Infinite repetition (be careful!)
let ones = repeat_iter(1)
// ones.collect() would run forever --- use .take() instead
print(ones.take(5).collect())  // [1, 1, 1, 1, 1]
```

Infinite Iterators

Calling `.collect()` on an infinite iterator will consume all available memory and hang your program. Always pair infinite sources with a limiting transformer like `.take()` before consuming.

26.3 Transformers: Reshaping the Stream

The real power of iterators emerges when you chain *transformers*—methods that wrap one iterator in another, building up a processing pipeline without doing any work until you ask for results.

26.3.1 .map() — Transform Each Element

The `.map()` method applies a closure to every element the iterator produces:

Listing 26.6: Mapping over an iterator

```
let prices = [19.99, 45.50, 12.00, 89.95]
let with_tax = iter(prices)
    .map(|price| { price * 1.08 })
    .collect()

print(with_tax)
// [21.5892, 49.14, 12.96, 97.146]
```

The closure receives each value and returns its transformation. Because `.map()` returns a new iterator, no work happens until `.collect()` pulls values through.

26.3.2 .filter() — Keep Only What Matters

The `.filter()` method takes a predicate closure and yields only the elements for which it returns `true`:

Listing 26.7: Filtering an iterator

```
let readings = [3.1, 7.8, 2.4, 9.0, 5.5, 1.2]
let high_readings = iter(readings)
    .filter(|r| { r > 5.0 })
    .collect()

print(high_readings)  // [7.8, 9.0, 5.5]
```

Internally, the filter iterator loops through the inner source, skipping elements that don't match, until it finds one that does—or the source is exhausted. You can see this logic in `src/iterator.c` where `iter_filter_next` calls the inner iterator in a loop.

26.3.3 .take() — Limit the Count

The `.take(n)` transformer yields at most `n` elements, then stops:

Listing 26.8: Taking the first few elements

```
let first_three = range_iter(0, 1000000)
    .take(3)
    .collect()

print(first_three)  // [0, 1, 2]
```

Despite the range spanning a million integers, only three are ever produced. This is laziness in action: the remaining 999,997 values are never computed.

26.3.4 .skip() — Jump Ahead

The companion to `.take()`, the `.skip(n)` transformer discards the first `n` elements and yields the rest:

Listing 26.9: Skipping elements

```
let after_header = iter(["name", "age", "city", "Alice", "30", "Portland"])
    .skip(3)
    .collect()

print(after_header)  // [Alice, 30, Portland]
```

You can combine `.skip()` and `.take()` to extract a window from a sequence:

Listing 26.10: Pagination with skip and take

```
let all_items = range_iter(1, 101)   // items 1 through 100
let page_size = 10
let page_number = 3

let page = all_items
    .skip((page_number - 1) * page_size)
    .take(page_size)
    .collect()

print(page)  // [21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
```

26.3.5 .enumerate() — Track the Index

When you need both the position and the value, `.enumerate()` wraps each element in a two-element array `[index, value]`:

Listing 26.11: Enumerating an iterator

```
let languages = ["Lattice", "Rust", "Python", "Go"]
let indexed = iter(languages).enumerate()

for pair in indexed.collect() {
    print(to_string(pair[0]) + ": " + pair[1])
}
// 0: Lattice
// 1: Rust
// 2: Python
// 3: Go
```

The index always starts at zero.

26.3.6 .zip() — Walk Two Iterators in Lockstep

The `.zip()` transformer pairs elements from two iterators into `[left, right]` arrays. It stops as soon as either iterator is exhausted:

Listing 26.12: Zipping two iterators

```
let names = iter(["Alice", "Bob", "Carol"])
let scores = iter([95, 87, 92])

let paired = names.zip(scores).collect()
print(paired)
// [[Alice, 95], [Bob, 87], [Carol, 92]]
```

Zip is especially useful for combining parallel data sources:

Listing 26.13: Merging labels with values

```
let labels = iter(["Temperature", "Humidity", "Pressure"])
let values = iter([72.1, 45.0, 1013.25])
let units = iter(["F", "%", "hPa"])

// Zip labels with values, then map to formatted strings
let report = labels.zip(values).zip(units)
    .map(|entry| {
        let label_value = entry[0]
        let unit = entry[1]
        label_value[0] + ": " + to_string(label_value[1]) + " " + unit
    })
    .collect()

for line in report {
    print(line)
}
// Temperature: 72.1 F
// Humidity: 45.0 %
// Pressure: 1013.25 hPa
```

26.3.7 Chaining Transformers

Transformers return iterators, so you can chain them into expressive pipelines:

Listing 26.14: A multi-step pipeline

```
let words = ["Lattice", "is", "a", "language", "for", "the", "curious"]

let result = iter(words)
    .filter(|w| { len(w) > 2 })
    .map(|w| { w + "!" })
    .enumerate()
    .take(3)
    .collect()

print(result)
// [[0, Lattice!], [1, language!], [2, for!]]
```

No intermediate arrays are created. Each value flows through the entire chain—from source to `.filter()` to `.map()` to `.enumerate()` to `.take()`—one at a time, on demand.

Reading Pipelines

Read iterator chains from top to bottom, like a recipe:

1. Start with the words.
2. Keep only words longer than two characters.
3. Append an exclamation mark.
4. Number them.
5. Take the first three.
6. Collect into an array.

26.4 Consumers: Triggering the Work

Transformers are lazy—they build up a chain of work but produce no output on their own. *Consumers* are the methods that actually pull values through the pipeline and produce a final result.

26.4.1 .collect() — Materialize into an Array

We have been using `.collect()` throughout this chapter. It drains the iterator and gathers every element into an array:

Listing 26.15: Collecting an iterator

```
let squares = range_iter(1, 6)
    .map(|n| { n * n })
    .collect()

print(squares)  // [1, 4, 9, 16, 25]
```

Under the hood, `iter_collect` in `src/iterator.c` starts with a small buffer (capacity 8) and doubles it as needed, appending each value from `iter_next` until the iterator signals done. The alias `.to_array()` does exactly the same thing.

26.4.2 .reduce() — Fold into a Single Value

The `.reduce()` method collapses an iterator into a single accumulated value. It takes a closure and an initial accumulator:

Listing 26.16: Reducing an iterator

```
let total = iter([10, 20, 30, 40])
    .reduce(|acc, val| { acc + val }, 0)

print(total)  // 100
```

The closure receives two arguments: the running accumulator and the current element. The second argument to `.reduce()` is the initial value for the accumulator.

Here is a more involved example—computing the maximum value:

Listing 26.17: Finding the maximum with reduce

```
let temperatures = [68.2, 71.5, 65.0, 73.8, 70.1]

let hottest = iter(temperatures)
    .reduce(|max_temp, temp| {
        if temp > max_temp { temp } else { max_temp }
    }, temperatures[0])

print(hottest)  // 73.8
```

26.4.3 .any() and .all() — Short-Circuit Boolean Tests

These two consumers answer yes-or-no questions about the sequence. Both *short-circuit*: they stop consuming as soon as the answer is determined.

Listing 26.18: Testing with .any() and .all()

```
let grades = [88, 92, 75, 95, 61]

let has_failing = iter(grades).any(|g| { g < 65 })
print(has_failing)  // true (stops after finding 61)

let all_passing = iter(grades).all(|g| { g >= 60 })
print(all_passing)  // true
```

The `.any()` consumer returns `true` as soon as it finds a matching element, without examining the rest. The `.all()` consumer returns `false` as soon as it finds a non-matching element. If the iterator is empty, `.any()` returns `false` and `.all()` returns `true`—consistent with the mathematical conventions for existential and universal quantification.

26.4.4 .count() — How Many?

The `.count()` consumer drains the iterator and returns the number of elements:

Listing 26.19: Counting elements

```
let long_words = iter(["cat", "elephant", "dog", "rhinoceros", "ox"])
    .filter(|w| { len(w) > 4 })
    .count()

print(long_words)  // 2
```

Consumers Are Destructive

Calling a consumer like `.collect()`, `.reduce()`, or `.count()` drains the iterator. If you try to use the same iterator afterwards, it will produce no more values. To reuse, build a fresh iterator or collect into an array first.

26.4.5 Summary of Consumers

Table 26.1: Iterator consumer methods

Method	Returns	Description
`.collect()`	`Array`	Gather all elements into an array
`.to_array()`	`Array`	Alias for `.collect()`
`.reduce(fn, init)`	`Any`	Fold with accumulator
`.any(predicate)`	`Bool`	True if any element matches
`.all(predicate)`	`Bool`	True if all elements match
`.count()`	`Int`	Count elements (consumes all)

26.5 Building Your Own Iterators

The built-in iterator methods cover the common cases, but sometimes you need a custom sequence—Fibonacci numbers, lines from a database cursor, or values generated by a complex algorithm.

Lattice's `lib/fn.lat` standard library module provides a protocol for building custom lazy sequences entirely in Lattice code. Import it, and you get access to a rich set of sequence constructors and transformers.

26.5.1 The Sequence Protocol

A lazy sequence in the `fn` module is a map with two keys:

- `"state"`: the current position or state of the iterator (any value).
- `"next"`: a function that takes the current state and returns a step map.

Each *step* is itself a map:

- `\{"value": v, "done": false, "state": next_state\}` — yields value `v` and advances.
- `\{"done": true\}` — the sequence is exhausted.

Lazy Sequence (fn module)

A lazy sequence threads state explicitly through each step. This design is compatible with Lattice's value semantics: no hidden mutable references, no shared state. The next function is pure—given the same state, it always produces the same step.

26.5.2 Building a Fibonacci Iterator

Let's build a lazy Fibonacci sequence using the `fn` module:

Listing 26.20: A custom Fibonacci iterator

```
import "lib/fn" as F

// State is [current, next_val]
let fibs = F.iterate([0, 1], |pair| {
    [pair[1], pair[0] + pair[1]]
})

// iterate() yields the state at each step,
// so we fmap to extract just the first element
let fib_numbers = F.fmap(fibs, |pair| { pair[0] })

// Take the first 10 Fibonacci numbers
let result = F.collect(F.take(fib_numbers, 10))
print(result)  // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

The key function here is `F.iterate(seed, f)`, which produces an infinite sequence: `seed`, `f(seed)`, `f(f(seed))`, and so on. We start with `[0, 1]` and advance by shifting the pair forward.

26.5.3 Custom Sequences from Scratch

You can also build a sequence manually by providing a state and a next function. Here is a sequence that yields powers of two:

Listing 26.21: Building a sequence from scratch

```
import "lib/fn" as F

fn powers_of_two() -> Map {
    // State is the current power
    let initial_state = 1
    let next_fn = |current| {
        let step = Map::new()
        step.set("value", current)
        step.set("done", false)
        step.set("state", current * 2)
        return step
    }

    let seq = Map::new()
    seq.set("state", initial_state)
    seq.set("next", next_fn)
    return seq
}

let result = F.collect(F.take(powers_of_two(), 8))
print(result)  // [1, 2, 4, 8, 16, 32, 64, 128]
```

The pattern is always the same: define an initial state, write a function that takes the current state and returns a step map, then combine them into a sequence map.

26.5.4 A Practical Example: Paginated API Results

Custom iterators shine when the data source is external or unbounded. Here is a sketch of a lazy sequence that fetches pages from an API:

Listing 26.22: Lazy pagination (conceptual)

```
import "lib/fn" as F

fn paginated_users(base_url: String) -> Map {
    // State is the current page number
    return F.iterate(1, |page| { page + 1 })
}

// In a real app, you would use fmap to fetch each page:
// let pages = F.fmap(paginated_users("/api/users"), |page| {
//     http_get(base_url + "?page=" + to_string(page))
// })
//
// Then take pages until the response is empty.
```

The lazy approach means you never fetch a page you don't need.

26.5.5 Composing fn Module Sequences

The `fn` module provides the same family of transformers as the built-in iterator methods, but as standalone functions that work with the sequence protocol:

Listing 26.23: Composing sequences with the fn module

```
import "lib/fn" as F

// Start with a range, filter, transform, and collect
let result = F.collect(
    F.take(
        F.fmap(
            F.select(F.range(1, 100), |x| { x % 3 == 0 }),
            |x| { x * x }
        ),
        5
    )
)

print(result)  // [9, 36, 81, 144, 225]
```

This computes the squares of the first five multiples of three. Every step is lazy—the range never produces 100 values, only as many as needed to fill the first five matches.

Table 26.2: fn module sequence functions

Function	Kind	Description
`F.range(start, end, step?)`	Source	Lazy integer range
`F.from_array(arr)`	Source	Lazy sequence from array
`F.repeat(value)`	Source	Infinite repetition
`F.iterate(seed, f)`	Source	Infinite `seed, f(seed), ...`
`F.fmap(seq, f)`	Transformer	Transform each element
`F.select(seq, pred)`	Transformer	Keep matching elements
`F.take(seq, n)`	Transformer	First `n` elements
`F.drop(seq, n)`	Transformer	Skip first `n`
`F.take_while(seq, pred)`	Transformer	Yield while predicate holds
`F.zip(seq1, seq2)`	Transformer	Pair elements from two sequences
`F.collect(seq)`	Consumer	Collect into array
`F.fold(seq, init, f)`	Consumer	Reduce with accumulator
`F.count(seq)`	Consumer	Count elements

Two Iterator Systems

Lattice has two complementary iteration systems:

1. **Built-in iterators** (`iter()`, `.map()`, `.filter()`, etc.) — implemented in C (`src/iterator.c`), accessed via method chaining. Best for everyday use.
2. **fn module sequences** (`F.range()`, `F.fmap()`, `F.select()`, etc.) — implemented in pure Lattice (`lib/fn.lat`), using a map-based protocol. Best for custom sequences and functional composition.

Both are lazy. The built-in system is faster (native C); the `fn` module is more flexible (user-extensible).

26.6 Why Laziness Matters

We have seen iterators in action, but let's make the case for *why* you should prefer them over eager array operations.

26.6.1 Memory Efficiency

Consider processing a large dataset:

Listing 26.24: Eager vs. lazy memory usage

```
// Eager: creates three intermediate arrays
let data = range_iter(0, 1000000).collect()  // 1M element array
let filtered = data.filter(|x| { x % 7 == 0 })  // another array
let mapped = filtered.map(|x| { x * x })  // yet another
let result = mapped[0..5]  // finally, what we wanted

// Lazy: creates zero intermediate arrays
let result_lazy = range_iter(0, 1000000)
    .filter(|x| { x % 7 == 0 })
    .map(|x| { x * x })
    .take(5)
    .collect()

print(result_lazy)  // [0, 49, 196, 441, 784]
```

The eager version allocates three full arrays before extracting five values. The lazy version allocates one small array (the final result) and produces only the 35 integers needed to fill it.

26.6.2 Computation Efficiency

Laziness also saves computation. When `.take(5)` has collected its five elements, the entire pipeline stops—no more filtering, no more mapping, and the range never produces its remaining elements.

This is especially valuable when the source is expensive (network calls, file I/O) or the filter is highly selective.

26.6.3 Infinite Sequences

Some sequences have no natural end—the natural numbers, a stream of sensor readings, lines from a log file that is still being written. Lazy iterators handle these gracefully because they never try to materialize the whole sequence:

Listing 26.25: Working with infinite sequences

```
import "lib/fn" as F

// The natural numbers
let naturals = F.range(1, 9999999)

// First 5 perfect squares
let squares = F.collect(
    F.take(
        F.fmap(naturals, |n| { n * n }),
        5
    )
)
print(squares)  // [1, 4, 9, 16, 25]

// First 5 numbers divisible by both 3 and 7
let div_3_and_7 = F.collect(
    F.take(
        F.select(F.range(1, 9999999), |n| { n % 3 == 0 && n % 7 == 0 }),
        5
    )
)
print(div_3_and_7)  // [21, 42, 63, 84, 105]
```

26.6.4 When to Stay Eager

Laziness is not always the right choice. If you need to access elements by index, sort the collection, or iterate over the data multiple times, collecting into an array first is the pragmatic approach. Iterators are single-pass and sequential—they trade random access for efficiency.

A Rule of Thumb

Use iterators when you are *flowing* data through a pipeline: read, transform, consume. Use arrays when you need to *hold* data: index, sort, search, iterate multiple times.

26.7 Under the Hood

If you are curious about how iterators work at the C level, here is a brief tour.

Each built-in iterator is a `LatValue` of type `VAL_ITERATOR`. It stores three things:

- A **state pointer** (`void *state`)—an opaque blob of data specific to the iterator kind (array index, range counter, wrapped inner iterator, etc.).
- A **next function** (`next_fn`)—a C function pointer that advances the state and returns the next value.
- A **free function** (`free_fn`)—cleans up the state when the iterator is no longer needed.

The `iter_next` call (defined in `include/iterator.h`) is a single function pointer dispatch:

```
return iter->as.iterator.next_fn(iter->as.iterator.state, &done);
```

Transformer iterators like `iter_take` wrap an inner iterator by storing it in their state struct (e.g., `IterTakeState` contains the inner iterator and a remaining count). When their `next_fn` is called, they delegate to the inner iterator's `next_fn`, applying their transformation. This forms a chain of function pointer calls—lightweight and allocation-free during iteration.

Iterator Ownership

Transformer constructors like `iter_take()` and `iter_map_transform()` *take ownership* of the inner iterator. Once you pass an iterator to a transformer, you should not use the original. The built-in method-chaining API handles this naturally—each `.take()` or `.map()` call consumes the receiver and returns a new iterator.

26.8 Exercises

1. **Sum of Squares.** Use `range_iter`, `.map()`, and `.reduce()` to compute the sum of squares of the integers from 1 to 100. What answer do you get?

2. **Custom Collatz Iterator.** The Collatz sequence for a starting number `n` is: if `n` is even, the next value is `n / 2`; if odd, it is `3 * n + 1`. The sequence ends when it reaches 1. Using the `fn` module's `iterate()` and `take_while()`, build a lazy Collatz sequence starting from 27 and collect it into an array. How many steps does it take?

3. **Zip and Reduce.** Given two arrays of equal length—one of item names and one of prices—use `iter()`, `.zip()`, and `.reduce()` to compute the total price. For example: names `["apple", "bread", "milk"]` and prices `[1.20, 3.50, 2.80]` should yield `7.50`.

4. **Infinite Primes.** Write a function that returns a lazy sequence (using the `fn` module) of prime numbers. Use it to print the first 20 primes. *Hint*: use `F.select` with a primality-testing closure over `F.range(2, 9999999)`.

5. **Interleave.** Write a function `interleave(seq1, seq2)` that takes two lazy sequences and produces a new sequence that alternates elements: first from `seq1`, then from `seq2`, then `seq1`, and so on. Stop when either sequence is exhausted.

What's Next Iterators give you a powerful way to express data transformations as pipelines. In the next chapter, we take this further: Chapter 27 introduces Lattice's functional programming tools—`pipe()`, `compose()`, currying, and the full `fn` standard library—that let you build elegant, composable abstractions on top of the iterator foundation we have laid here.

Chapter 27

Functional Programming in Lattice

Lattice is not a purely functional language—it has mutable variables, side effects, and imperative loops. But it provides a rich set of functional tools that, when used well, can make your code more concise, more composable, and easier to reason about.

In Chapter 26, we built pipelines of iterator transformations. This chapter takes those ideas further. We will explore Lattice's built-in functional primitives—`pipe()`, `compose()`, and `identity()`—then dive into the `fn` standard library's currying, partial application, and collection utilities. By the end, you will know when functional style shines in Lattice and when to reach for something else.

27.1 pipe(), compose(), and identity()

These three built-in functions form the foundation of functional composition in Lattice. They are implemented directly in the runtime (`src/runtime.c` and `src/eval.c`), so they are always available without any imports.

27.1.1 pipe() — Threading Values Through Functions

The `pipe()` function takes a value and one or more functions, then threads the value through each function left to right:

Listing 27.1: Basic pipe usage

```
let result = pipe(5,
    |x| { x * 2 },
    |x| { x + 1 },
    |x| { x * x }
)
print(result)  // 121  (5 -> 10 -> 11 -> 121)
```

Each function receives the output of the previous one. The first argument is the initial value; every subsequent argument must be a closure.

pipe()

`pipe(val, f1, f2, ..., fn)` computes `fn(...f2(f1(val)))`. It evaluates left to right: the value flows *forward* through the chain, like water through a series of pipes.

Without `pipe()`, you would write deeply nested function calls:

Listing 27.2: Nested calls vs. pipe

```
// Without pipe: read inside-out
let result = square(increment(double(5)))

// With pipe: read top-to-bottom
let result = pipe(5,
    |x| { x * 2 },
    |x| { x + 1 },
    |x| { x * x }
)
```

The pipe version reads in the order that operations happen—no mental stack unwinding required.

Here is a more realistic example, processing a string:

Listing 27.3: String processing with pipe

```
let raw_input = "  Hello, Lattice World!  "

let processed = pipe(raw_input,
    |s| { s.trim() },
    |s| { s.lower() },
    |s| { s.replace(",", "") },
    |s| { s.split(" ") }
)

print(processed)  // [hello, lattice, world!]
```

pipe() with Named Functions

You are not limited to anonymous closures. Named functions work too:

```
fn double(x: Int) -> Int { return x * 2 }
fn increment(x: Int) -> Int { return x + 1 }

let result = pipe(5, double, increment)
print(result)  // 11
```

27.1.2 compose() — Building New Functions from Old

While `pipe()` immediately applies a chain of functions to a value, `compose()` builds a *new function* from two existing ones—without calling either:

Listing 27.4: Composing functions

```
let double = |x| { x * 2 }
let increment = |x| { x + 1 }

// compose(f, g) returns a new function: |x| { f(g(x)) }
let double_then_increment = compose(increment, double)

print(double_then_increment(5))   // 11  (double(5)=10, increment(10)=11)
print(double_then_increment(10))  // 21  (double(10)=20, increment(20)=21)
```

Argument Order

`compose(f, g)` applies `g` first, then `f`. In mathematical notation: `compose(f, g)(x) = f(g(x))`. This is right-to-left, like function composition in mathematics. If you find this confusing, use `pipe()` instead—it flows left-to-right.

The value of `compose()` is that it creates a reusable function you can pass around, store in variables, or use as a callback:

Listing 27.5: Composed functions as callbacks

```
let to_celsius = |f| { (f - 32) * 5 / 9 }
let round_temp = |c| { to_int(c) }
let format_temp = |c| { to_string(c) + " C" }

// Build a single conversion function
let convert = compose(format_temp, compose(round_temp, to_celsius))

let temps_f = [98.6, 212.0, 32.0, 72.5]
let temps_c = iter(temps_f).map(convert).collect()
print(temps_c)  // [37 C, 100 C, 0 C, 22 C]
```

Under the hood, `compose()` creates a new closure that captures both `f` and `g` as upvalues. In the tree-walk evaluator (`src/eval.c`), it builds an AST node for `f(g(x))` with a fresh environment. In the stack VM (`src/runtime.c`), it generates bytecode: load `g`, call with `x`, load `f`, call with the result. Either way, the composed function behaves like any other closure.

27.1.3 identity() — The Do-Nothing Function

The `identity()` function returns its argument unchanged:

Listing 27.6: The identity function

```
print(identity(42))        // 42
print(identity("hello"))   // hello
print(identity([1, 2, 3])) // [1, 2, 3]
```

At first glance, this seems useless. Why would you want a function that does nothing? The answer is that `identity()` serves as a *default* or *neutral element* in functional pipelines:

Listing 27.7: identity() as a default transform

```
fn process_data(values: Array, transform: Fn) -> Array {
    return iter(values).map(transform).collect()
}

// Sometimes we want a transform, sometimes we don't
let raw = [1, 2, 3, 4, 5]

let doubled = process_data(raw, |x| { x * 2 })
print(doubled)  // [2, 4, 6, 8, 10]

let unchanged = process_data(raw, identity)
print(unchanged)  // [1, 2, 3, 4, 5]
```

Rather than adding special-case logic for "no transformation," you pass `identity` as the transform. The code stays uniform.

Another use: `identity` is the neutral element for `compose()`. Composing any function with `identity` gives back the original function:

Listing 27.8: identity() as composition neutral

```
let double = |x| { x * 2 }

let same = compose(double, identity)
print(same(5))  // 10
```

27.2 The fn Standard Library Module

The `fn` module (`lib/fn.lat`) bundles a collection of functional programming utilities into a single import. We encountered its lazy sequence functions in Chapter 26; now let's explore the rest.

Listing 27.9: Importing the fn module

```
import "lib/fn" as F
```

27.2.1 Function Composition Utilities

The module provides several utilities for building and combining functions:

Listing 27.10: comp, flip, and constant

```
import "lib/fn" as F

// comp(f, g) is a module-accessible wrapper around compose()
let add_one_then_double = F.comp(|x| { x * 2 }, |x| { x + 1 })
print(add_one_then_double(4))  // 10

// flip(f) reverses the argument order of a 2-argument function
let div = |a, b| { a / b }
let inv_div = F.flip(div)
print(inv_div(2, 10))  // 5  (calls div(10, 2))

// constant(v) returns a function that always returns v
let always_zero = F.constant(0)
print(always_zero(99))     // 0
print(always_zero("hi"))   // 0
```

27.2.2 apply_n — Repeated Application

The `F.apply_n(f, n, value)` function applies `f` to `value` a total of `n` times:

Listing 27.11: Repeated function application

```
import "lib/fn" as F

let double = |x| { x * 2 }
print(F.apply_n(double, 0, 3))  // 3
print(F.apply_n(double, 1, 3))  // 6
print(F.apply_n(double, 4, 1))  // 16  (1 -> 2 -> 4 -> 8 -> 16)
```

This is a controlled form of recursion—useful for simulations, numerical methods, or any situation where you want to iterate a transformation a fixed number of times.

27.2.3 thread() — Module-Level Piping

The `F.thread(value, f1, f2)` function is the module's equivalent of a two-function `pipe()`, for cases where you want to use the `fn` module's qualified syntax:

Listing 27.12: thread for 2-step piping

```
import "lib/fn" as F

let result = F.thread(10,
    |x| { x + 5 },
    |x| { x * 2 }
)
print(result)  // 30
```

For longer chains, the built-in `pipe()` is more flexible since it accepts any number of functions.

27.2.4 Collection Utilities

Beyond sequences and composition, the `fn` module includes utilities for working with arrays and maps.

group_by — Categorize Elements

Listing 27.13: Grouping elements

```
import "lib/fn" as F

let words = ["apple", "avocado", "banana", "blueberry", "cherry"]
let grouped = F.group_by(words, |w| { w[0] })

print(grouped)
// {a: [apple, avocado], b: [banana, blueberry], c: [cherry]}
```

The key function determines which group each element belongs to. The result is a map from keys to arrays of matching elements.

partition — Split into Two Groups

Listing 27.14: Partitioning an array

```
import "lib/fn" as F

let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let result = F.partition(numbers, |n| { n % 2 == 0 })

print(result[0])  // [2, 4, 6, 8, 10]  (matches)
print(result[1])  // [1, 3, 5, 7, 9]   (non-matches)
```

frequencies — Count Occurrences

Listing 27.15: Counting element frequencies

```
import "lib/fn" as F

let votes = ["yes", "no", "yes", "yes", "no", "abstain"]
let tally = F.frequencies(votes)

print(tally)
// {yes: 3, no: 2, abstain: 1}
```

uniq_by — Deduplicate by Key

Listing 27.16: Deduplication by a key function

```
import "lib/fn" as F

let items = [1, 2, 3, 4, 5, 6, 7, 8, 9]
let unique_mod3 = F.uniq_by(items, |x| { x % 3 })

print(unique_mod3)  // [1, 2, 3]  (first occurrence of each mod-3 class)
```

27.2.5 The Result Type

The `fn` module also provides a lightweight `Result` type for representing success or failure without exceptions:

Listing 27.17: The Result type from the fn module

```
import "lib/fn" as F

let success = F.ok(42)
let failure = F.err("something went wrong")

print(F.is_ok(success))   // true
print(F.is_err(failure))  // true

print(F.unwrap(success))  // 42
print(F.unwrap_or(failure, 0))  // 0

// Transform the Ok value
let doubled = F.map_result(success, |x| { x * 2 })
print(F.unwrap(doubled))  // 84

// Chain operations that might fail
let chained = F.flat_map_result(success, |x| {
    if x > 0 { F.ok(x * 10) } else { F.err("must be positive") }
})
print(F.unwrap(chained))  // 420
```

The `F.try_fn()` wrapper catches runtime errors and returns a Result:

Listing 27.18: Catching errors with try_fn

```
import "lib/fn" as F

let safe = F.try_fn(|_| { 100 / 4 })
print(F.unwrap(safe))  // 25

let risky = F.try_fn(|_| { 100 / 0 })
print(F.is_err(risky))  // true
```

27.3 Currying and Partial Application Patterns

Currying and partial application are techniques for breaking a multi-argument function into a chain of single-argument functions. This lets you "pre-fill" some arguments and pass the specialized function around.

27.3.1 curry() — One Argument at a Time

The `F.curry()` function transforms a 2- or 3-argument function into a chain of single-argument functions:

Listing 27.19: Currying a 2-argument function

```
import "lib/fn" as F

let add = F.curry(|a, b| { a + b })

// Now add(5) returns a new function that adds 5
let add_five = add(5)
print(add_five(3))   // 8
print(add_five(10))  // 15

// Or call it all at once via chaining
print(add(2)(3))     // 5
```

Three-argument functions work the same way:

Listing 27.20: Currying a 3-argument function

```
import "lib/fn" as F

let volume = F.curry(|length, width, height| {
    length * width * height
})

let slab = volume(10)        // fix the length
let narrow_slab = slab(2)    // fix the width
print(narrow_slab(5))        // 100

print(volume(3)(4)(5))       // 60
```

Currying

Currying transforms a function `f(a, b)` into `f(a)(b)`. Each call with one argument returns a new function that "remembers" the provided argument and waits for the rest.

27.3.2 partial() — Bind Some Arguments Now

While currying requires you to provide arguments one at a time in order, `F.partial()` lets you bind any number of leading arguments at once:

Listing 27.21: Partial application

```
import "lib/fn" as F

let greet = |greeting, name| { greeting + ", " + name + "!" }

let hello = F.partial(greet, "Hello")
let howdy = F.partial(greet, "Howdy")

print(hello("Alice"))  // Hello, Alice!
print(howdy("Bob"))    // Howdy, Bob!
```

You can bind multiple arguments at once:

Listing 27.22: Binding multiple arguments

```
import "lib/fn" as F

let log_message = |level, component, msg| {
    print("[" + level + "] " + component + ": " + msg)
}

let log_error = F.partial(log_message, "ERROR")
let log_db_error = F.partial(log_message, "ERROR", "database")

log_error("auth", "invalid token")
// [ERROR] auth: invalid token

log_db_error("connection timeout")
// [ERROR] database: connection timeout
```

27.3.3 Currying with Iterators

Currying and partial application become especially powerful when combined with iterators, because they let you create specialized transform functions on the fly:

Listing 27.23: Curried functions in pipelines

```
import "lib/fn" as F

// A curried multiplier
let multiply = F.curry(|factor, x| { factor * x })

let prices = [12.50, 8.99, 24.00, 5.75]

// Create specialized multipliers
let with_tax = multiply(1.08)
let discounted = multiply(0.85)

// Use them in pipelines
let taxed_prices = iter(prices).map(with_tax).collect()
print(taxed_prices)  // [13.5, 9.7092, 25.92, 6.21]

let sale_prices = iter(prices).map(discounted).collect()
print(sale_prices)  // [10.625, 7.6415, 20.4, 4.8875]
```

Compare this with the alternative—writing a fresh closure for each operation. Currying lets you express the intent ("multiply by 1.08") more directly than `|x| { x * 1.08 }`.

Curry or Partial?

Use `curry()` when you want to apply arguments one at a time and the function has 2–3 parameters. Use `partial()` when you want to bind one or more leading arguments in a single call, especially for functions with many parameters.

27.4 When Functional Style Shines in Lattice

Functional programming is a tool, not a religion. Lattice gives you both imperative and functional styles; the art is knowing when each one serves you best.

27.4.1 Data Transformation Pipelines

Functional style is at its best when you are transforming data through a series of well-defined steps:

Listing 27.24: A data transformation pipeline

```
import "lib/fn" as F

let raw_records = [
    "Alice,95,Math",
    "Bob,87,Science",
    "Carol,92,Math",
    "Dave,78,Science",
    "Eve,96,Math"
]

// Parse, filter, and summarize in one flow
let top_math_scores = pipe(raw_records,
    |records| { iter(records).map(|r| { r.split(",") }).collect() },
    |parsed| { iter(parsed).filter(|r| { r[2] == "Math" }).collect() },
    |math| { iter(math).map(|r| { to_int(r[1]) }).collect() },
    |scores| { iter(scores).filter(|s| { s >= 90 }).collect() }
)

print(top_math_scores)  // [95, 92, 96]
```

Each step has a clear purpose, and the whole pipeline reads like a recipe.

27.4.2 Callback-Heavy APIs

When working with APIs that accept callbacks—event handlers, sorting comparators, validation rules—functional tools help you build specialized callbacks without repetitive boilerplate:

Listing 27.25: Building validators with partial application

```
import "lib/fn" as F

let is_between = |min_val, max_val, x| {
    x >= min_val && x <= max_val
}

let is_valid_age = F.partial(is_between, 0, 150)
let is_valid_score = F.partial(is_between, 0, 100)

let ages = [25, -3, 42, 200, 17]
let valid_ages = iter(ages).filter(is_valid_age).collect()
print(valid_ages)  // [25, 42, 17]
```

27.4.3 Configuration and Strategy Patterns

Functions are values in Lattice. You can store them in maps, pass them as arguments, and return them from other functions. This makes the strategy pattern natural:

Listing 27.26: Strategy pattern with functions

```
let strategies = Map::new()
strategies.set("uppercase", |s| { s.upper() })
strategies.set("lowercase", |s| { s.lower() })
strategies.set("reverse", |s| { s.reverse() })

fn apply_strategy(strategy_name: String, text: String) -> String {
    let transform = strategies.get(strategy_name)
    return transform(text)
}

print(apply_strategy("uppercase", "hello"))   // HELLO
print(apply_strategy("reverse", "Lattice"))   // ecittaL
```

27.4.4 When Imperative Style Is Better

Functional style is not always the clearest choice. Prefer imperative code when:

- The logic involves complex branching or early returns that don't fit neatly into a pipeline.

- You need to accumulate state across multiple collections simultaneously.
- The code would require deeply nested function calls that obscure the intent.
- Performance matters and you need fine-grained control over allocation.

Listing 27.27: Sometimes a loop is clearer

```
// Functional version -- hard to follow
let result = pipe(items,
    |xs| { iter(xs).filter(|x| { x > threshold }).collect() },
    |xs| {
        iter(xs).reduce(|acc, x| {
            if x > acc { x } else { acc }
        }, 0)
    }
)

// Imperative version -- immediately clear
flux max_val = 0
for item in items {
    if item > threshold && item > max_val {
        max_val = item
    }
}
```

The best Lattice code uses both styles where each is strongest. Functional pipelines for data transformations; imperative loops for stateful logic.

The Litmus Test

If you have to read a functional pipeline more than twice to understand what it does, it is probably trying to be too clever. Break it into named intermediate variables or switch to a loop. Clarity always beats cleverness.

27.5 Putting It All Together

Let's close with a larger example that combines iterators, `pipe()`, currying, and the `fn` module into a cohesive program.

Listing 27.28: A complete functional example

```
import "lib/fn" as F

// --- Data ---
let inventory = [
    ["Widget A", 29.99, 150],
    ["Widget B", 9.99, 500],
    ["Gadget C", 49.95, 30],
    ["Gadget D", 14.50, 200],
    ["Doohickey E", 4.99, 1000]
]

// --- Helpers built with currying ---
let min_price = F.curry(|threshold, item| { item[1] >= threshold })
let min_stock = F.curry(|threshold, item| { item[2] >= threshold })

// --- Pipeline: find valuable items with good stock ---
let report = pipe(inventory,
    |items| { iter(items).filter(min_price(10.0)).collect() },
    |items| { iter(items).filter(min_stock(100)).collect() },
    |items| {
        iter(items).map(|item| {
            let name = item[0]
            let total_value = item[1] * item[2]
            name + ": $" + to_string(total_value)
        }).collect()
    }
)

for line in report {
    print(line)
}
// Widget A: $4498.5
// Widget B: $4995.0
// Gadget D: $2900.0
```

Each piece is small and testable on its own. The curried predicates can be reused across different pipelines. The `pipe()` call makes the overall flow readable. And because the iterator operations are lazy where possible, no unnecessary work is done.

27.6 Exercises

1. **Compose Three.** Lattice's built-in `compose()` takes exactly two functions. Write a function `compose3(f, g, h)` that composes three: `compose3(f, g, h)(x) = f(g(h(x)))`. Test it by composing three arithmetic operations.

2. **Curried Filter.** Using `F.curry()`, create a reusable `longer_than` predicate that filters strings by minimum length. Use it to extract all words longer than 5 characters from the sentence `"The quick brown fox jumped over the lazy dog"`.

3. **Word Frequency Pipeline.** Given a block of text (a multi-line string), use `pipe()` and the `fn` module to:

   (a) Split the text into words.
   (b) Convert all words to lowercase.
   (c) Count the frequency of each word using `F.frequencies()`.

   Print the resulting frequency map.

4. **Result Pipeline.** Write a sequence of operations using `F.ok()`, `F.flat_map_result()`, and `F.map_result()` to:

   (a) Start with a string that might represent a number.
   (b) Parse it to an integer (returning `F.err()` if it fails).
   (c) Double the number.
   (d) Check that it is positive (returning `F.err()` if not).

   Test with both valid and invalid inputs.

What's Next With iterators and functional tools in your kit, you are well-equipped to write expressive, composable Lattice code. In Chapter 28, we turn to a different kind of composition: organizing your code into *modules* and *packages*, so that the functions and types you build can be shared, reused, and managed as your projects grow.

Part VIII

Modules, Packages, and Project Structure

Chapter 28

Modules and Imports

As your programs grow beyond a single file, you face a fundamental organizational question: how do you split code into manageable pieces without losing the ability to share functionality between them? Most languages answer this with a module system, and Lattice is no different—except that Lattice's approach is deliberately minimal. There are no package declarations at the top of each file, no directory-based namespaces, no visibility modifiers beyond a single keyword. A file is a module. An `export` makes something public. An `import` brings it into scope.

Let's see how it works.

28.1 Your First Import

Suppose you have a utility file called `math_utils.lat` that defines a couple of functions:

Listing 28.1: math_utils.lat — a utility module

```
fn add(x: Int, y: Int) -> Int {
    return x + y
}

fn sub(x: Int, y: Int) -> Int {
    return x - y
}

let PI = 3.14159
```

Notice that this file has no `export` keywords. We will talk about what that means in a moment. For now, the important thing is that every top-level binding—`add`, `sub`, and `PI`—is visible to importers.

To use these from another file, you write:

Listing 28.2: main.lat — importing a module

```
import "math_utils" as math

print(math.add(2, 3))   // 5
print(math.sub(10, 4))  // 6
print(math.PI)           // 3.14159
```

The `import "math_utils" as math` statement does three things:

1. Locates the file `math_utils.lat` (appending the `.lat` extension automatically).
2. Executes that file's top-level code in an isolated scope.
3. Bundles the module's exported bindings into a map and assigns it to the name `math`.

You then access individual members with dot notation: `math.add`, `math.PI`, and so on.

File Extensions Are Optional

When you write `import "math_utils"`, Lattice automatically appends `.lat` if the path does not already end with it. You can also write `import "math_utils.lat"` explicitly—both forms resolve to the same file.

28.2 Import Styles

Lattice supports two import syntaxes, each suited to different situations.

28.2.1 Aliased Import: `import ... as`

The form we already saw brings in the entire module under a name you choose:

Listing 28.3: Aliased import

```
import "geometry/shapes" as shapes

let circle = shapes.make_circle(5.0)
let area = shapes.circle_area(circle)
```

This is the preferred style when you use many things from a module, or when you want the reader to see *where* each symbol comes from. The alias can be anything you like—`import "geometry/shapes" as geo` works too.

28.2.2 Selective Import: `import {...} from`

When you only need a few specific symbols, you can pull them directly into your local scope:

Listing 28.4: Selective import

```
import { add, PI } from "math_utils"

print(add(2, 3))  // 5
print(PI)          // 3.14159
```

Each name listed inside the braces becomes a local binding. You do not prefix them with a module name—`add` is `add`, not `math.add`.

This style is great when you need a handful of specific functions and want to keep your code concise. The trade-off is that the reader cannot immediately tell which module a function came from without scrolling back to the import.

Missing Exports Are Runtime Errors

If you try to selectively import a name that the module does not export, Lattice throws an error at runtime. For example, if `math_utils.lat` does not define `multiply`, then `import \{ multiply \} from "math_utils"` will fail with:

```
module 'math_utils' does not export 'multiply'
```

The compiler generates a runtime check for each selectively imported name, validating that the field exists in the module map before binding it.

28.2.3 Bare Import

There is a third form you will see less often: a bare import with no alias and no selective names.

Listing 28.5: Bare import — side effects only

```
import "setup"
```

This executes the module's top-level code but discards the resulting module map. It is useful for modules that exist purely for their side effects—initializing a global configuration, registering plugins, or running setup code.

Module

A *module* in Lattice is a single `.lat` source file. When imported, the file is compiled and executed in an isolated scope. The module's top-level bindings (functions, values, structs, enums) are collected into a map that the importer can access. There is no separate "module declaration" syntax—the file *is* the module.

28.3 The Export System

So far, our example modules have had no `export` keywords. In that case, Lattice uses *legacy mode*: every top-level binding is exported. This is convenient for small scripts and quick experiments, but for real projects you want explicit control over your module's public API.

28.3.1 Explicit Exports

Add the `export` keyword before any top-level declaration to mark it as public:

Listing 28.6: export_explicit.lat — explicit exports

```
export fn add(a: Int, b: Int) -> Int {
    return a + b
}

export let PI = 3.14159

fn internal_helper() -> Int {
    42
}

let secret = "hidden"
```

In this module, only `add` and `PI` are visible to importers. The function `internal_helper` and the value `secret` are private—they exist inside the module for its own use, but they do not appear in the module map.

Legacy vs. Explicit Export Mode

When the parser encounters at least one `export` keyword in a file, it switches to *explicit mode*: only names marked with `export` are included in the module map. If a file contains *no* `export` keywords at all, it operates in *legacy mode*, where all top-level bindings are exported.
This two-mode system lets quick scripts "work" without ceremony, while larger projects get proper encapsulation.

The `export` keyword can precede:

- Functions: `export fn calculate(...)`
- Variables: `export let MAX_SIZE = 256` or `export fix THRESHOLD = 100`
- Structs: `export struct Point { ... }`
- Enums: `export enum Color { ... }`
- Traits: `export trait Drawable { ... }`

28.3.2 Exporting Structs and Enums

When you export a struct, importers can create instances of it and use its constructor function. When you export an enum, importers can access its variants:

Listing 28.7: geometry.lat — exporting structs and enums

```
export struct Point {
    x: Int,
    y: Int
}

export fn make_point(x: Int, y: Int) -> Point {
    return Point { x: x, y: y }
}

export enum Direction { North, South, East, West }
```

Listing 28.8: Using exported structs and enums

```
import { Point, make_point, Direction } from "geometry"

let origin = make_point(0, 0)
print(origin.x)  // 0

let heading = Direction::North
```

28.3.3 What Gets Filtered Out

Regardless of export mode, Lattice always hides two categories of names:

1. Names starting with double underscores (`__`)—these are internal metadata used by the runtime.
2. Names containing a colon (`:`)—these are namespaced internal keys used for struct method dispatch and trait implementations.

You can see this filtering logic in `src/ast.c`, in the `module_should_export` function. It runs on every name in the module scope before adding it to the module map.

28.3.4 Re-exporting from Other Modules

A common pattern in larger projects is to create a "facade" module that imports from several internal modules and re-exports a curated public API:

Listing 28.9: api.lat — re-exporting a clean interface

```
import { double, BASE_CONST } from "internal/base_utils"

export fn double_it(x: Int) -> Int {
    return double(x)
}

export let REEXPORTED_CONST = BASE_CONST
```

The consumer of `api.lat` sees only `double_it` and `REEXPORTED_CONST`. They never need to know about `internal/base_utils.lat` at all.

28.4 Built-in Modules

Lattice ships with a set of built-in modules that are always available, regardless of where your script lives or whether you have any `lat_modules/` directory. These are implemented directly in the runtime (you can find the registration logic in `src/runtime.c`) and are loaded without touching the filesystem.

Here are the built-in modules:

Table 28.1: Lattice's built-in modules

Module	Description
`math`	Mathematical functions and constants
`fs`	File system operations (read, write, stat, mkdir)
`path`	Path manipulation (join, basename, dirname, extension)
`json`	JSON parsing and serialization
`toml`	TOML parsing and serialization
`yaml`	YAML parsing and serialization
`crypto`	Cryptographic hashing (SHA-256, MD5, Base64)
`http`	HTTP client (GET, POST, PUT, DELETE)
`net`	Low-level networking (TCP sockets, DNS)
`os`	Operating system interface (env vars, exec, platform)
`time`	Time and date functions
`regex`	Regular expression matching
`progress`	Terminal progress bars

To use a built-in module, import it by its bare name:

Listing 28.10: Using built-in modules

```
import "math" as math
import "json" as json
import "fs" as fs

let angle = math.PI / 4.0
print(math.sin(angle))  // 0.7071...

let data = json.parse("{\"name\": \"Lattice\"}")
print(data["name"])  // Lattice

let contents = fs.read("config.toml")
print(contents)
```

You can also use the selective import form with built-in modules:

Listing 28.11: Selective import from built-in modules

```
import { sin, cos, PI } from "math"

let x = cos(PI / 3.0)
let y = sin(PI / 3.0)
print("x=${x}, y=${y}")
```

How Built-in Modules Are Detected

When the VM encounters an `import` statement, it first checks whether the path is a bare name (no directory separators, no leading dot). If so, it calls `rt_try_builtin_import()` to look up the name in a table of built-in module factories. If the name matches, the module map is constructed in C and returned directly—no file is read from disk. Only if this check fails does Lattice proceed to file-based module resolution. You can trace this logic in `src/stackvm.c`, in the `OP_IMPORT` handler.

28.5 Module Resolution

When you write `import "something"`, how does Lattice find the right file? The resolution algorithm follows a predictable sequence. Understanding it will save you from mysterious "module not found" errors.

28.5.1 The Resolution Order

For a given import path, Lattice tries these strategies in order:

1. **Built-in modules.** If the path is a bare name (like `"math"` or `"json"`), check the built-in module table. If found, return immediately.

2. **Package resolution.** If the path is still a bare name and step 1 did not match, try the `lat_modules/` directory. This looks for the module in several locations (detailed below).

3. **Relative file path.** Treat the path as a file path relative to the importing script's directory. Append `.lat` if the path does not already end with it. Resolve to an absolute path using `realpath()`.

28.5.2 Package Resolution Strategies

When resolving a bare module name through `lat_modules/`, Lattice tries four paths in order (implemented in `pkg_resolve_module()` in `src/package.c`):

1. `lat_modules/⟨name⟩/main.lat` — a package with a standard entry point.
2. `lat_modules/⟨name⟩/src/main.lat` — a package using a `src/` subdirectory.
3. The `entry` field from `lat_modules/⟨name⟩/lattice.toml` — a package with a custom entry point defined in its manifest.
4. `lat_modules/⟨name⟩.lat` — a single-file package.

If the project directory does not yield a result, Lattice also tries these same four strategies relative to the current working directory.

Listing 28.12: Importing a third-party package

```
// If you have lat_modules/leftpad/main.lat installed:
import "leftpad" as lp

let padded = lp.pad("hello", 10)
print(padded)  // "     hello"
```

28.5.3 Relative and Nested Imports

For your own project files, use relative paths:

Listing 28.13: Relative path imports

```
// Import a file in the same directory
import "utils" as utils

// Import a file in a subdirectory
import "models/user" as user_model

// Import a file in a parent directory
import "../shared/config" as config
```

Paths are resolved relative to the directory of the file containing the import statement, not relative to where you ran `clat`.

Use Consistent Project Structure

A good convention is to keep your project's main entry point at the root and organize supporting modules in subdirectories:

```
my_project/
  main.lat
  models/
    user.lat
    post.lat
  utils/
    validation.lat
    formatting.lat
  lattice.toml
```

Then your imports read naturally: `import "models/user" as user`, `import "utils/validation" as valid`.

28.5.4 Module Caching

Lattice caches modules by their *resolved absolute path*. If two different files both import `"utils"`, the module is compiled and executed only once. Subsequent imports receive a deep clone of the cached module map.

This has two important consequences:

1. **Side effects run once.** If a module's top-level code prints a message or opens a file, that happens during the first import. Later imports skip execution entirely and use the cached result.

2. **Each importer gets its own copy.** Because the cache returns a deep clone, modifying a value obtained from a module does not affect other importers. The phase system still applies—if you import a frozen value, you get a frozen copy.

Listing 28.14: Module caching in action

```
// counter.lat
let count = 0
print("counter module loaded!")  // printed only once

export fn get_count() -> Int {
    return count
}
```

Listing 28.15: Two files importing the same module

```
// file_a.lat
import "counter" as c
// prints: "counter module loaded!"
print(c.get_count())  // 0

// file_b.lat (if imported after file_a.lat in the same program)
import "counter" as c
// nothing printed --- cached
print(c.get_count())  // 0
```

28.6 Organizing Multi-File Projects

Now that we understand the mechanics, let's look at how to organize a real project.

28.6.1 A Minimal Project

The smallest structured Lattice project looks like this:

```
my_app/
  lattice.toml
  main.lat
```

The `lattice.toml` file declares your project's metadata and dependencies (we will cover it in detail in the next chapter). The `main.lat` file is the entry point.

28.6.2 Growing Beyond One File

As the project grows, extract cohesive functionality into modules:

```
weather_station/
  lattice.toml
  main.lat
  sensors/
    temperature.lat
    humidity.lat
    pressure.lat
  alerts/
    thresholds.lat
    notifier.lat
  utils/
    formatting.lat
    logging.lat
```

Listing 28.16: weather_station/main.lat — tying it all together

```
import "sensors/temperature" as temp
import "sensors/humidity" as humid
import "alerts/thresholds" as thresh
import "alerts/notifier" as notify
import "utils/formatting" as fmt

let reading = temp.read_sensor()
let humidity = humid.read_sensor()

if reading > thresh.HIGH_TEMP {
    notify.send_alert(fmt.format_reading(reading, "F"))
}

print(fmt.format_reading(reading, "F"))
print(fmt.format_reading(humidity, "%"))
```

Listing 28.17: sensors/temperature.lat

```
export fn read_sensor() -> Float {
    // In a real application, this would read from hardware
    return 72.4
}

export fn to_celsius(fahrenheit: Float) -> Float {
    return (fahrenheit - 32.0) * 5.0 / 9.0
}
```

Listing 28.18: alerts/thresholds.lat

```
export let HIGH_TEMP = 100.0
export let LOW_TEMP = 32.0
export let HIGH_HUMIDITY = 80.0
export let LOW_HUMIDITY = 20.0
```

28.6.3 Facade Modules

When a subsystem has many internal modules, create a facade that presents a clean interface:

Listing 28.19: sensors/mod.lat — a facade module

```
import { read_sensor, to_celsius } from "sensors/temperature"
import { read_sensor } from "sensors/humidity"
import { read_sensor } from "sensors/pressure"

// Re-export a curated API
export fn read_temperature() -> Float {
    return read_sensor()
}

export fn read_temperature_celsius() -> Float {
    return to_celsius(read_sensor())
}
```

No Implicit Index Files

Unlike Node.js, Lattice does not automatically look for an `index.lat` or `mod.lat` when you import a directory name. If you write `import "sensors"`, Lattice looks for `sensors.lat`, not `sensors/index.lat`. To use a facade, import it explicitly: `import "sensors/mod" as sensors`.

28.6.4 Namespacing with Aliased Imports

When two modules export the same name, aliased imports prevent collisions:

Listing 28.20: Avoiding name collisions

```
import "models/user" as user_mod
import "models/admin" as admin_mod

// Both modules export make_record, but there is no conflict
let regular = user_mod.make_record("alice", "alice@example.com")
let admin = admin_mod.make_record("bob", "admin")
```

With selective imports, you would have a conflict—two different `make_record` bindings fighting for the same name. Aliased imports sidestep this entirely.

28.6.5 Scoped Imports

Import statements are not limited to the top of a file. You can import inside a function body, which makes the module available only within that scope:

Listing 28.21: Scoped import inside a function

```
fn generate_report(data: Array) -> String {
    import "json" as json
    import "utils/formatting" as fmt

    let formatted = fmt.tabulate(data)
    return json.stringify(formatted)
}
// json and fmt are not visible here
```

Scoped imports are compiled to local variable definitions rather than globals. The module is still cached globally, so there is no performance penalty for importing inside a function that gets called many times.

28.7 Under the Hood: How Imports Work

For readers who want to understand the machinery, here is what happens when Lattice encounters an import statement.

28.7.1 Parsing

The parser (in `src/parser.c`) recognizes two syntactic forms and creates a `STMT_IMPORT` AST node. The node stores three fields:

- `module_path` — the string literal from the import (always present).
- `alias` — the identifier after `as` (null if not provided).
- `selective_names` — an array of identifiers from the `\{...\}` list (null for aliased or bare imports).

28.7.2 Compilation

The compiler (`src/stackcompiler.c`) turns the AST node into bytecode. The core instruction is `OP_IMPORT`, which takes a single operand: the index of the module path in the chunk's constant pool.

For an aliased import like `import "utils" as u`:

1. Emit `OP_IMPORT` with the path constant index.
2. Emit `OP_DEFINE_GLOBAL` (or add a local, if inside a function) with the alias name.

For a selective import like `import \{ add, PI \} from "math_utils"`, the compiler generates a more elaborate sequence for each name:

1. Emit `OP_DUP` to copy the module map on the stack.
2. Emit `OP_GET_FIELD` to extract the named export.
3. Emit `OP_DUP` and `OP_JUMP_IF_NOT_NIL` to validate the export exists.
4. If nil, emit `OP_THROW` with an error message like `"module 'math_utils' does not export 'multiply'"`.
5. Otherwise, emit `OP_DEFINE_GLOBAL` (or add local) for that name.

After processing all selective names, a final `OP_POP` discards the module map from the stack.

28.7.3 Runtime Execution

When the VM hits `OP_IMPORT` at runtime (in `src/stackvm.c`), it:

1. **Reads the path** from the constant pool.
2. **Checks built-in modules** via `rt_try_builtin_import()`. If the path matches a stdlib module, the C implementation constructs a module map and pushes it onto the stack. Done.
3. **Resolves the file path.** Tries `pkg_resolve_module()` for bare names (checking `lat_modules/`), then falls back to appending `.lat` and resolving relative to the script directory.
4. **Checks the cache.** The VM maintains a hash map (`vm->module_cache`) keyed by absolute path. If the module has been loaded before, a deep clone of the cached module map is pushed and execution continues.
5. **Reads and compiles the source file.** The source is lexed, parsed, and compiled into a module chunk. Module chunks are compiled without an auto-call to `main()`.
6. **Executes in an isolated scope.** The VM pushes a new scope with `env_push_scope()`, runs the module's bytecode, then pops the scope. This ensures the module's internal variables do not leak into the caller's environment.
7. **Builds the module map.** After execution, the VM iterates over the module scope's bindings. Each binding passes through `module_should_export()` to determine visibility. Exported bindings are deep-cloned into a new map.
8. **Caches and returns.** The module map is stored in the cache and pushed onto the stack for the caller to use.

Closures Capture the Module Environment

When a module exports a function, that function is a closure that captures the module's environment. This means exported functions can call other functions defined in the same module—even ones that are not exported. The VM copies all module bindings into the base scope before popping the module's scope, ensuring that closure environments remain valid.

28.8 Exercises

1. **Split and import.** Take any program you have written so far that is longer than 50 lines. Extract at least two cohesive groups of functions into separate module files. Use explicit `export` to control what each module exposes. Verify that the program still works after the split.
2. **Selective vs. aliased.** Write a module called `colors.lat` that exports at least five functions (`to_hex`, `to_rgb`, `lighten`, `darken`, `mix`). Write two consumer files: one that uses an aliased import and another that uses selective imports. Which style do you prefer for this module? Why?
3. **The facade pattern.** Create a `database/` directory with three modules: `connection.lat`, `query.lat`, and `migration.lat`. Each should export at least two functions. Then create

`database/mod.lat` that re-exports a curated subset of the combined API. Write a `main.lat` that imports only from the facade.

4. **Built-in module explorer.** Write a program that imports the `math`, `json`, and `os` built-in modules. Use `math.sin` and `math.cos` to compute points on a unit circle, serialize them to JSON with `json.stringify`, and print the current platform with `os.platform()`.

5. **Module caching investigation.** Create a module called `counter.lat` that prints `"loaded!"` at the top level and exports a function. Import it from two different modules in the same program. Verify that `"loaded!"` is printed only once. Then modify one of the importers to use the module inside a function body (scoped import). Does the caching behavior change?

What's Next

Now that we can split code across files and control what each module exposes, the next question is: how do we manage *other people's* code? In Chapter 29, we will explore Lattice's built-in package manager—`clat init`, `lattice.toml`, semantic versioning, dependency resolution, and the `lat_modules/` directory that we've already been referencing. The module system you just learned is the foundation; the package manager builds the ecosystem on top of it.

Chapter 29

The Package Manager

In the previous chapter, we learned how to split our own code across modules and import them. But programming does not happen in isolation. Eventually, you will want to use a library someone else wrote—a JSON schema validator, a CLI argument parser, a color formatting toolkit. That is where the package manager comes in.

Lattice's package manager is built directly into `clat`, the same binary you use to run programs. There is no separate tool to install, no Node.js-style global package manager that updates independently from the language. Everything lives in one place: your manifest, your lock file, and a `lat_modules/` directory that works much like you would expect if you have used `npm` or `cargo`.

Let's start by creating a project.

29.1 Initializing a Project with `clat init`

Every managed Lattice project begins with a `lattice.toml` file. You can create one by hand, but the fastest way is:

```
mkdir weather-app && cd weather-app
clat init
```

Output:

```
Created lattice.toml
```

Open the generated file and you will see:

Listing 29.1: A freshly generated lattice.toml

```
[package]
name = "weather-app"
version = "0.1.0"
```

Three things to notice:

1. The `name` is derived from the current directory name.
2. The `version` starts at `0.1.0`, following semantic versioning conventions.
3. There is no explicit `entry` field—when omitted, Lattice defaults to `main.lat` as the entry point.

If you run `clat init` in a directory that already has a `lattice.toml`, it refuses and prints an error. It will never overwrite an existing manifest.

Custom Entry Points

If your project's main file is not called `main.lat`, add an `entry` field to the `[package]` section:

```
[package]
name = "weather-app"
version = "0.1.0"
entry = "src/app.lat"
```

This tells both the package manager and importers where to find the project's primary module.

29.2 The `lattice.toml` Manifest

The `lattice.toml` file is the single source of truth for your project's metadata and dependencies. It uses TOML syntax—the same format Rust uses for `Cargo.toml` and Python uses for `pyproject.toml`.

Here is a complete manifest for a project with dependencies:

Listing 29.2: A lattice.toml with metadata and dependencies

```
[package]
name = "weather-app"
version = "1.2.0"
description = "A weather monitoring application"
license = "MIT"
entry = "main.lat"

[dependencies]
http-client = "^1.0.0"
json-schema = "~0.3.2"
cli-args = "2.1.0"
color-fmt = "*"
```

29.2.1 The [package] Section

The [package] table contains your project's metadata:

Table 29.1: Fields in the [package] section

Field	Required	Description
name	Yes	The package name (derived from directory by clat init)
version	Yes	Semantic version string (e.g., "1.2.3")
description	No	A short description of the package
license	No	SPDX license identifier (e.g., "MIT", "Apache-2.0")
entry	No	Entry point file (defaults to "main.lat")

29.2.2 The [dependencies] Section

The [dependencies] table maps package names to version constraints. Each key is a package name, and each value is a string specifying which versions are acceptable:

```
[dependencies]
http-client = "^1.0.0"
json-schema = "~0.3.2"
cli-args = "2.1.0"
color-fmt = "*"
```

We will break down what those version strings mean in Section 29.4.

29.3 Adding, Removing, and Installing Packages

The package manager provides three commands for managing dependencies.

29.3.1 `clat add`

To add a dependency to your project:

```
clat add http-client ^1.0.0
```

Output:

```
Added http-client@^1.0.0 to dependencies
  Installing http-client@^1.0.0... ok

All dependencies installed.
```

This does two things:

1. Adds `http-client = "^1.0.0"` to the `[dependencies]` section of your `lattice.toml`.
2. Runs `clat install` to fetch and install the package.

If you omit the version constraint, it defaults to `"*"` (any version):

```
clat add color-fmt
```

If the package is already a dependency, `clat add` updates its version constraint instead of adding a duplicate.

No `lattice.toml` Yet?

If you run `clat add` before running `clat init`, the package manager creates a minimal `lattice.toml` for you automatically, deriving the package name from the current directory. You do not need a separate initialization step if you start by adding dependencies.

29.3.2 `clat install`

To install all dependencies listed in your manifest:

```
clat install
```

Output:

```
  Installing http-client@^1.0.0... ok
  Installing json-schema@~0.3.2... ok
  Installing cli-args@2.1.0... ok
  Installing color-fmt@*... ok

All dependencies installed.
```

For each dependency, the installer follows this resolution strategy:

1. **Check `lat_modules/`.** If the package directory already exists, verify that the installed version satisfies the constraint. If it does, skip the download.
2. **Check the lock file.** If `lattice.lock` exists, prefer the locked version (as long as it still satisfies the manifest constraint). This ensures reproducible builds.
3. **Check the local file registry.** If the `LATTICE_REGISTRY` environment variable is set to a `file://` URL, try copying the package from that directory.
4. **Fetch from the HTTP registry.** Query the registry for available versions, find the best match, download it to the global cache, and copy it into `lat_modules/`.

After all dependencies are installed, `clat install` builds a dependency graph and checks for circular dependencies. If a cycle is detected, it reports the full cycle path and fails:

```
error: circular dependency detected: alpha -> beta -> gamma -> alpha
```

Finally, it writes (or updates) the `lattice.lock` file.

29.3.3 `clat remove`

To remove a dependency:

```
clat remove color-fmt
```

Output:

```
Removed color-fmt
```

This does three things:

1. Removes the package from `lattice.toml`.
2. Deletes `lat_modules/color-fmt/` from disk.
3. Removes the entry from `lattice.lock`.

If the package is not listed as a dependency, `clat remove` prints an error and does nothing.

29.4 Semantic Versioning and Dependency Resolution

Lattice uses semantic versioning (semver) for all version strings. A version has three numeric components:

Semantic Version

A version string `MAJOR.MINOR.PATCH` communicates intent:

- **MAJOR** — incremented for breaking, incompatible changes.
- **MINOR** — incremented for backward-compatible new features.
- **PATCH** — incremented for backward-compatible bug fixes.

For example, `2.3.1` means major version 2, minor version 3, patch level 1.

29.4.1 Version Constraints

When you list a dependency in `lattice.toml`, the version string is a *constraint*, not an exact version. Lattice supports six constraint types:

29.4.2 Caret vs. Tilde

The most commonly used constraints are caret (`^`) and tilde (`~`). Here is how they differ in practice:

Table 29.2: Semver constraint types

Syntax	Name	Meaning
`"*"`	Wildcard	Any version. Useful during early development.
`"1.2.3"`	Exact	Exactly version 1.2.3 and nothing else.
`"^1.2.3"`	Caret (Compatible)	Same major version, at least 1.2.3. Matches `1.2.3`, `1.3.0`, `1.99.0`, but not `2.0.0`.
`"~1.2.3"`	Tilde (Approximate)	Same major *and* minor version, at least patch 3. Matches `1.2.3`, `1.2.9`, but not `1.3.0`.
`">=1.2.3"`	Minimum	Version 1.2.3 or newer. No upper bound.
`"<=1.2.3"`	Maximum	Version 1.2.3 or older. No lower bound.

Listing 29.3: Caret vs. tilde constraints

```
# Caret: ^1.2.3
#   Allows: 1.2.3, 1.2.4, 1.3.0, 1.99.0
#   Blocks: 2.0.0 (different major)

# Tilde: ~1.2.3
#   Allows: 1.2.3, 1.2.4, 1.2.99
#   Blocks: 1.3.0 (different minor)
```

When to use caret: You trust the package author to maintain backward compatibility within a major version. This is the default choice for most dependencies.

When to use tilde: You want only bug fixes for a specific feature set. This is more conservative and useful when a minor version introduced a behavior you depend on.

Default to Caret

If you are unsure which constraint to use, start with caret (`^`). It gives package authors room to ship new features without breaking your code, while still protecting you from major-version breaking changes.

29.4.3 How Resolution Works

When `clat install` needs to resolve a version constraint, it follows this process:

1. **Query the registry** for all published versions of the package. The registry endpoint is `GET /packages/<name>/versions`, which returns a JSON array.

2. **Filter** the version list against the constraint using `pkg_semver_satisfies()` (implemented in `src/package.c`).
3. **Select the highest** matching version. Lattice always prefers the newest compatible version, following the common "maximal version" strategy.

For example, if the registry has versions `[1.0.0, 1.1.0, 1.2.0, 2.0.0]` and your constraint is `^1.0.0`, the resolver picks `1.2.0`—the highest version that shares the same major version.

29.5 The Lock File and Reproducible Builds

Every time `clat install` succeeds, it writes a `lattice.lock` file. This file records the *exact* versions that were installed, so that future installs produce identical results.

29.5.1 What the Lock File Looks Like

Listing 29.4: A typical lattice.lock

```
# This file is auto-generated by clat. Do not edit manually.

[[package]]
name = "http-client"
version = "1.2.0"
source = "registry"
checksum = ""

[[package]]
name = "json-schema"
version = "0.3.5"
source = "registry"
checksum = ""

[[package]]
name = "cli-args"
version = "2.1.0"
source = "local"
checksum = ""
```

Each `[[package]]` entry records:

- **`name`** — the package name.
- **`version`** — the exact resolved version.

- **`source`** — where the package came from: `"registry"` (HTTP registry), `"local"` (already in `lat_modules/`), or `"path"` (file-based registry).
- **`checksum`** — a SHA-256 hash for integrity verification (reserved for future use).

29.5.2 How the Lock File Ensures Reproducibility

When `clat install` finds an existing `lattice.lock`, it prefers the locked version for each dependency—*as long as it still satisfies the constraint in* `lattice.toml`. This means:

- **Same machine, next week:** Running `clat install` again installs the same versions, even if newer versions have been published.
- **Different machine, same lock file:** A teammate clones your repository, runs `clat install`, and gets the exact same dependency versions.
- **Updating the constraint:** If you change a version constraint in `lattice.toml` so that the locked version no longer satisfies it, the next `clat install` fetches a new version and updates the lock file.

Commit Your Lock File

Always commit `lattice.lock` to version control. This is how you guarantee reproducible builds across machines and over time. The `lat_modules/` directory, on the other hand, should be in your `.gitignore`—it can always be reconstructed from the lock file.

29.5.3 When Lock Entries Are Updated

The lock file is regenerated in these situations:

1. `clat install` runs and resolves new or changed dependencies.
2. `clat add` adds a new dependency and triggers an install.
3. `clat remove` removes a dependency and strips its entry from the lock file.

29.6 The Package Registry and Cache

29.6.1 The Lattice Registry

Lattice's default package registry lives at:

```
https://registry.lattice-lang.org/v1
```

The package manager communicates with the registry using two HTTP endpoints:

Table 29.3: Registry API endpoints

Endpoint	Purpose
`GET /packages/⟨name⟩/versions`	List available versions (JSON)
`GET /packages/⟨name⟩/⟨version⟩`	Download package source

The versions endpoint returns JSON like `\{"versions":["1.0.0","1.1.0","2.0.0"]\}`. The download endpoint returns either a single `main.lat` file or a TOML bundle, depending on the package.

29.6.2 Overriding the Registry

You can point the package manager at a different registry using the `LATTICE_REGISTRY` environment variable:

```
# Use a private registry
export LATTICE_REGISTRY=https://packages.mycompany.com/v1
clat install

# Use a local directory as a registry (great for testing)
export LATTICE_REGISTRY=file:///home/user/my-packages
clat install
```

The `file://` protocol is particularly useful for development. If you set `LATTICE_REGISTRY=file:///path/to/packages`, the package manager copies packages directly from that directory into `lat_modules/` without any HTTP requests.

29.6.3 The Global Package Cache

When a package is downloaded from the registry, it is stored in a global cache before being copied into your project's `lat_modules/`:

```
~/.lattice/packages/<name>/<version>/
```

For example, `http-client` version `1.2.0` would be cached at:

```
~/.lattice/packages/http-client/1.2.0/
```

The next time any project on your machine needs that same package at that same version, the package manager copies it from the cache instead of downloading it again. You will see the message `(cached http-client@1.2.0)` in the install output when this happens.

Clearing the Cache

If you ever need to force a fresh download, delete the cached package directory:

```
rm -rf ~/.lattice/packages/http-client/1.2.0
clat install
```

Or clear the entire cache:

```
rm -rf ~/.lattice/packages
```

29.6.4 The `lat_modules/` Directory

After installation, your project's dependency tree lives in `lat_modules/`:

```
my-project/
  lattice.toml
  lattice.lock
  main.lat
  lat_modules/
    http-client/
      lattice.toml
      main.lat
    json-schema/
      lattice.toml
      src/
        main.lat
    cli-args/
      lattice.toml
      main.lat
```

Each package directory contains at minimum a `lattice.toml` and one or more `.lat` source files. The module resolution system (described in Section 28.5) knows how to find the entry point for each package.

To use an installed package in your code, import it by its package name:

Listing 29.5: Importing installed packages

```
import "http-client" as http
import { validate } from "json-schema"
import "cli-args" as args

let response = http.get("https://api.weather.gov/points/44.9,-93.2")
let data = validate(response.body, schema)
```

The import system tries the built-in modules first, then falls back to `lat_modules/`—exactly as described in Section 28.5.

29.7 Dependency Graphs and Cycle Detection

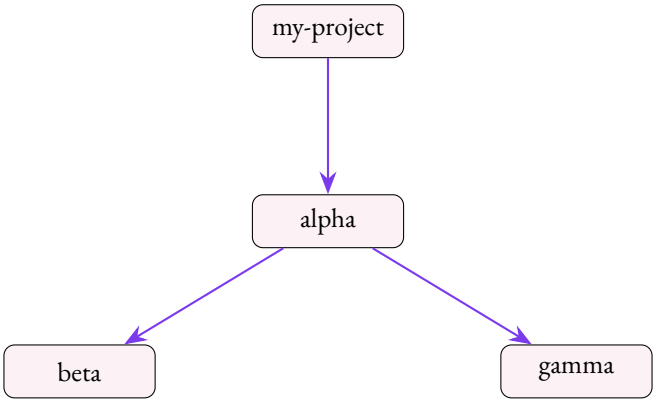
When packages depend on other packages, those dependencies form a directed graph. Lattice builds this graph after installation and checks it for cycles.

29.7.1 How the Graph Is Built

The dependency graph is constructed by reading each installed package's `lattice.toml`:

1. Your project is the root node.
2. Each of your direct dependencies becomes a node with an edge from the root.
3. For each dependency, the package manager reads *its* `lattice.toml` and adds edges for its dependencies.
4. This continues via breadth-first search until all transitive dependencies have been discovered.

For example, if your project depends on `alpha`, and `alpha` depends on `beta` and `gamma`, the graph looks like:



29.7.2 Cycle Detection

Circular dependencies—where `A` depends on `B`, `B` depends on `C`, and `C` depends on `A`—are a serious problem. They make it impossible to determine a safe installation and loading order.

Lattice detects cycles using a depth-first search (DFS) algorithm. Each node is colored during traversal:

- **White**: not yet visited.
- **Gray**: currently being explored (on the DFS stack).
- **Black**: fully explored (all descendants processed).

If the DFS encounters a gray node—meaning we have followed a path back to a node that is still being explored—a cycle has been found. The algorithm then traces back through the DFS stack to produce a human-readable cycle path:

```
error: circular dependency detected: alpha -> beta -> gamma -> alpha
```

You can see this implementation in `src/package.c`, in the `dfs_visit()` and `pkg_dep_graph_has_cycle()` functions.

Circular Dependencies Are Fatal

If a circular dependency is detected, `clat install` fails and does not write a lock file. You must resolve the cycle—typically by refactoring one of the packages to remove the circular relationship—before installation can succeed.

29.8 Under the Hood: The Install Pipeline

For readers who want the full picture, here is what happens when you run `clat install`, step by step.

1. **Read `lattice.toml`.** Parse the TOML into a `PkgManifest` structure containing the project metadata and dependency list. If the file does not exist, print an error suggesting `clat init`.

2. **Check for an existing lock file.** If `lattice.lock` exists, parse it into a `PkgLock` structure. The locked versions will be preferred during resolution.

3. **Create `lat_modules/`.** If the directory does not exist, create it.

4. **Resolve each dependency.** For each entry in `[dependencies]`:

   - If the lock file has an entry for this package, and the locked version satisfies the manifest constraint, use the locked version.
   - Check if `lat_modules/⟨name⟩` already exists. If so, verify its version.
   - If not installed: try a `file://` registry (if configured), then fall back to the HTTP registry.
   - On the HTTP registry: query available versions, select the best match, check the global cache (`~/.lattice/packages/`), download if needed, copy to `lat_modules/`.

5. **Build the dependency graph.** Read each installed package's `lattice.toml` to discover transitive dependencies. Construct a directed graph using BFS.

6. **Check for cycles.** Run DFS-based cycle detection on the graph. If a cycle is found, report it and abort.

7. **Write `lattice.lock`.** Serialize the resolved versions, sources, and checksums to TOML.

Flat Installation

Lattice uses a *flat* dependency layout: all packages live directly under `lat_modules/`, not nested inside each other. This means that if two packages depend on the same third package, there is only one copy. The trade-off is that version conflicts between transitive dependencies are not yet automatically resolved—this is a known limitation that will be addressed in a future version of the package manager.

29.9 Putting It All Together

Let's walk through a complete workflow from project creation to running with dependencies.

Listing 29.6: Complete project workflow

```
# Create a new project
mkdir sensor-dashboard && cd sensor-dashboard
clat init

# Add dependencies
clat add json-utils ^1.0.0
clat add http-server ^0.5.0
clat add template-engine ~2.1.0
```

Your `lattice.toml` now looks like:

```
[package]
name = "sensor-dashboard"
version = "0.1.0"

[dependencies]
json-utils = "^1.0.0"
http-server = "^0.5.0"
template-engine = "~2.1.0"
```

Write your application:

Listing 29.7: main.lat — using installed packages

```
import "json-utils" as json
import "http-server" as server
import { render } from "template-engine"

import "sensors/temperature" as temp
import "sensors/humidity" as humid

fn handle_dashboard(req: any) -> any {
    let readings = [
        temp.read_sensor(),
        humid.read_sensor()
    ]
    let body = render("dashboard.html", json.stringify(readings))
    return server.response(200, body)
}

let app = server.create(8080)
server.route(app, "/", handle_dashboard)
server.start(app)
print("Dashboard running on http://localhost:8080")
```

Notice how third-party packages (from `lat_modules/`) and your own modules (from `sensors/`) are imported with the same syntax. The module system handles the resolution transparently.

Run the program:

```
clat main.lat
```

Share the project with a colleague:

```
# In .gitignore:
lat_modules/

# Commit everything else:
git add lattice.toml lattice.lock main.lat sensors/
git commit -m "Initial sensor dashboard"
git push
```

When your colleague clones the repository, they run:

```
clat install
clat main.lat
```

The lock file ensures they get the exact same dependency versions you used.

29.10 Exercises

1. **Initialize and explore.** Run `clat init` in a new directory. Examine the generated `lattice.toml`. Try running `clat init` again—what happens? Now manually add a `description` and `license` field to the manifest.
2. **Semver reasoning.** Given the available versions `[1.0.0, 1.1.0, 1.2.3, 1.3.0, 2.0.0, 2.1.0]`, which version does each constraint select?
   - `"*"`
   - `"^1.1.0"`
   - `"~1.2.0"`
   - `">=1.3.0"`
   - `"<=1.2.3"`
   - `"1.2.3"`
3. **Local file registry.** Create a directory called `my-registry` containing a subdirectory `greetings` with a `main.lat` that exports a `hello` function. Set `LATTICE_REGISTRY=file:///path/to/my-registry` and run `clat add greetings`. Verify that the package appears in `lat_modules/` and can be imported.
4. **Lock file investigation.** Add a dependency with a caret constraint. Examine the generated `lattice.lock`. Change the version constraint in `lattice.toml` to a tilde constraint for the same minor version. Run `clat install` again. Does the lock file change? Why or why not?
5. **Circular dependency detection.** Create three packages (`alpha`, `beta`, `gamma`) in a local file registry. Make `alpha` depend on `beta`, `beta` depend on `gamma`, and `gamma` depend on `alpha`. Create a project that depends on `alpha` and run `clat install`. Observe the error message. Then fix the cycle by removing one of the dependency edges and verify that the install succeeds.

What's Next

With modules, imports, and packages under your belt, you have everything you need to build and share Lattice projects of any size. But writing code is only half the story—maintaining it is the other

half. In Chapter 30, we will explore Lattice's built-in code formatter and documentation generator, tools that help keep your codebase consistent and well-documented as it grows.

Part IX

Tooling and Developer Experience

Chapter 30

The Formatter and Doc Generator

Every codebase drifts. One developer uses tabs, another uses spaces. Someone wraps function arguments at 80 columns; someone else at 120. Doc comments get written in three different styles. A week later, the "git blame" for every file reads like a United Nations assembly. Lattice solves this with two built-in tools: `clat fmt` for formatting source code and `clat doc` for generating documentation. Both are first-class citizens of the `clat` CLI—no third-party installation required.

30.1 `clat fmt` — Formatting Your Code

Let's start with a messy file. Suppose a colleague has handed us `sensor.lat` with inconsistent spacing, missing indentation, and some creative use of whitespace:

Listing 30.1: Before formatting: messy whitespace and indentation

```
fn read_temperature(  sensor_id:Int,unit: String   ) -> Float{
    let raw=sensor_id *   0.48+2.1
if unit ==  "celsius"{
return raw
}else{
    return raw*9.0/5.0+32.0
}}
```

One command brings order to the chaos:

Listing 30.2: Terminal: running clat fmt

```
// In your terminal:
// $ clat fmt sensor.lat
```

After formatting, the file is rewritten in place:

Listing 30.3: After formatting: clean and consistent

```
fn read_temperature(sensor_id: Int, unit: String) -> Float {
    let raw = sensor_id * 0.48 + 2.1
    if unit == "celsius" {
        return raw
    } else {
        return raw * 9.0 / 5.0 + 32.0
    }
}
```

Notice what the formatter did:

- Normalized spacing around operators (`=`, `*`, `+`) to exactly one space on each side.
- Enforced the *attached brace style*: the opening `{` appears on the same line as the `fn`, `if`, and `else` keywords, preceded by a single space.
- Inserted a space after the colon in type annotations (`sensor_id: Int`) and after commas.
- Applied consistent four-space indentation at each nesting level.
- Placed the `else` clause on the same line as the closing brace of the `if` block (`} else {`).
- Ensured a single trailing newline at the end of the file.

clat fmt

The `clat fmt` command reads a Lattice source file, reformats it using the canonical style, and writes the result back. It operates on a character-by-character scan of the source (implemented in `src/formatter.c`), preserving all comments and string contents while normalizing whitespace and indentation. The default indentation width is four spaces, and the default target line width is 100 columns.

30.1.1 Formatting Multiple Files

You can format an entire project in one go by passing multiple files:

Listing 30.4: Formatting several files at once

```
// $ clat fmt src/main.lat src/utils.lat lib/math.lat
```

30.1.2 Check Mode: CI-Friendly Verification

In a continuous integration pipeline, you don't want to *rewrite* files—you want to *verify* that they're already formatted. The `--check` flag does exactly this:

Listing 30.5: Using --check in CI

```
// $ clat fmt --check src/main.lat
// If already formatted: exits with code 0
// If not: exits with code 1 and prints a diff
```

Under the hood, `--check` calls the `lat_format_check()` function (see `include/formatter.h`), which formats the source and compares the result to the original. If they match, the file is already canonical.

Add Formatting to Your CI Pipeline

A common pattern for Lattice projects:

```
// In your CI configuration:
// $ clat fmt --check src/ lib/
```

This ensures every pull request follows the same style. No more bikeshedding in code reviews about where the braces go.

30.1.3 Reading from Standard Input

The formatter can also read from standard input with the `--stdin` flag. This is useful for editor integrations and piped workflows:

Listing 30.6: Formatting via stdin

```
// $ cat messy.lat | clat fmt --stdin
// Formatted output goes to stdout
```

The `lat_format_stdin()` function in `src/formatter.c` reads all of standard input into a buffer, formats it, and writes the result to standard output.

30.2 `--width` for Configurable Line Width

The default line width target is 100 columns—a comfortable middle ground for modern screens. But preferences vary. Some teams work on laptops with split panes and prefer 80 columns. Others have ultrawide monitors and 120 feels right.

The `--width` flag lets you choose:

Listing 30.7: Setting line width to 80 columns

```
// $ clat fmt --width 80 src/main.lat
```

Listing 30.8: Setting line width to 120 columns

```
// $ clat fmt --width 120 src/main.lat
```

When the formatter encounters a line that exceeds the target width, it makes wrapping decisions—for instance, breaking long function parameter lists across multiple lines. Passing zero (or omitting the flag) uses the default of 100, as defined by the `DEFAULT_LINE_WIDTH` constant in `src/formatter.c`.

Width is a Target, Not a Hard Limit

The `--width` value is a *target*, not an absolute ceiling. Some constructs—like a long string literal or a deeply nested expression—cannot be broken further without changing program semantics. In those cases, the formatter will exceed the target rather than produce incorrect code.

Let's see the width flag in action. Consider a function with many parameters:

Listing 30.9: A function with many parameters

```
fn create_report(title: String, author: String, created_at: String, tags: Array, body:
    String, draft: Bool) -> Map {
    let report = Map::new()
    report["title"] = title
    report["author"] = author
    report["created_at"] = created_at
    report["tags"] = tags
    report["body"] = body
    report["draft"] = draft
    return report
}
```

At `--width 80`, the formatter may wrap the parameter list to keep lines within bounds. At `--width 120`, the entire signature might fit on a single line.

Team Convention

Pick a width for your project and stick with it. Add it to your project's build script or Makefile:

```
// Makefile target:
// fmt:
//     clat fmt --width 100 src/ lib/
```

30.3 /// Doc Comments

Lattice uses triple-slash (`///`) comments for documentation. Unlike regular `//` comments, doc comments are extracted by the documentation generator and attached to the declaration that follows them.

Listing 30.10: Doc comments on a function

```
/// Calculate the body mass index for a patient.
/// Returns the BMI as a Float value.
fn calculate_bmi(weight_kg: Float, height_m: Float) -> Float {
    return weight_kg / (height_m * height_m)
}
```

Doc comments can span multiple consecutive lines. The `///` prefix and one optional space after it are stripped, and the remaining text becomes the documentation body.

30.3.1 Documenting All Declaration Types

Doc comments work on every kind of top-level declaration: functions, structs, enums, traits, impl blocks, and even variables.

Listing 30.11: Doc comments on structs and enums

```
/// A patient record in the healthcare system.
/// Contains demographic and medical information.
struct Patient {
    /// The patient's full legal name.
    name: String,
    /// Age in years.
    age: Int,
    /// Blood type (e.g., "A+", "O-").
    blood_type: String
}

/// Possible outcomes of a medical test.
enum TestResult {
    /// The test came back normal.
    Negative,
    /// The test detected the condition.
    Positive,
    /// The sample was insufficient or contaminated.
    Inconclusive
}
```

Notice that each struct field and enum variant can have its own doc comment. The documentation generator (covered in the next section) extracts these and associates them with the correct field or variant.

30.3.2 Module-Level Doc Comments

If a `///` comment block appears at the very top of a file—before any declaration—it is treated as a *module-level* doc comment. This is the place to describe what the file or module does:

Listing 30.12: Module-level documentation

```
/// The geometry module provides functions for calculating
/// areas and perimeters of common shapes.
///
/// All functions accept dimensions in the same unit
/// and return results in squared units where applicable.

fn circle_area(radius: Float) -> Float {
    return 3.14159265 * radius * radius
}

fn rectangle_area(width: Float, height: Float) -> Float {
    return width * height
}
```

The doc extractor in `src/doc_gen.c` distinguishes module-level docs from declaration docs by checking whether the doc block is followed by a declaration keyword (`fn`, `struct`, `enum`, `trait`, `impl`, `flux`, `fix`, `let`, or `export`). If not, it's treated as the module description.

30.3.3 Doc Comment Conventions

While Lattice doesn't enforce a specific doc comment format, the community has settled on a few conventions:

Listing 30.13: Recommended doc comment style

```
/// Open a database connection to the given host.
///
/// The connection uses a pooled TCP socket under the hood.
/// If the host is unreachable, returns an error after
/// the specified timeout.
///
/// Parameters are validated before connecting. An empty
/// host string will produce an immediate error.
fn connect(host: String, port: Int, timeout_ms: Int) -> Map {
    // ... implementation
    return Map::new()
}
```

- First line: a brief one-sentence summary.

- Blank line (an empty `///` line) separating the summary from the longer description.

- Additional paragraphs for context, usage notes, or caveats.

Regular Comments Are Ignored

Only `///` comments are picked up by `clat doc`. Regular `//` comments and block comments (`/* ... */`) are treated as implementation notes and excluded from generated documentation. If you write documentation with `//` instead of `///`, it won't appear in the output.

30.4 `clat doc` — Generating Documentation

The `clat doc` command extracts doc comments from one or more `.lat` files and renders them as human-readable documentation.

Listing 30.14: Basic usage of clat doc

```
// $ clat doc geometry.lat
```

By default, this prints Markdown to standard output. Let's walk through a complete example. Suppose we have `math_utils.lat`:

Listing 30.15: A well-documented utility module

```
/// Utility functions for common mathematical operations.

/// Clamp a value to the range [low, high].
/// If the value is below low, returns low.
/// If above high, returns high.
fn clamp(value: Float, low: Float, high: Float) -> Float {
    if value < low {
        return low
    }
    if value > high {
        return high
    }
    return value
}

/// Linear interpolation between two values.
/// When t is 0.0, returns a. When t is 1.0, returns b.
fn lerp(a: Float, b: Float, t: Float) -> Float {
    return a + (b - a) * t
}

/// A 2D point in Cartesian space.
struct Point {
    /// Horizontal coordinate.
    x: Float,
    /// Vertical coordinate.
    y: Float
}

/// Compute the Euclidean distance between two points.
fn distance(p1: Point, p2: Point) -> Float {
    let dx = p2.x - p1.x
    let dy = p2.y - p1.y
    return (dx * dx + dy * dy) ** 0.5
}
```

Running `clat doc math_utils.lat` produces:

Listing 30.16: Generated Markdown output (abbreviated)

```
// Utility functions for common mathematical operations.
//
// ## Functions
//
// ### `clamp(value: Float, low: Float, high: Float) -> Float`
//
// Clamp a value to the range [low, high].
// If the value is below low, returns low.
// If above high, returns high.
//
// ### `lerp(a: Float, b: Float, t: Float) -> Float`
//
// Linear interpolation between two values.
// When t is 0.0, returns a. When t is 1.0, returns b.
//
// ## Structs
//
// ### `struct Point`
//
// A 2D point in Cartesian space.
//
// | Field | Type    | Description            |
// |-------|---------|------------------------|
// | `x`   | `Float` | Horizontal coordinate. |
// | `y`   | `Float` | Vertical coordinate.   |
```

The documentation generator groups items by kind: functions, structs, enums, traits, implementations, and variables. Each section includes the full signature and associated documentation.

30.4.1 Documenting a Directory

Pass a directory to document all `.1at` files inside it:

Listing 30.17: Documenting an entire directory

```
// $ clat doc src/
```

When processing multiple files, each file gets its own heading in the output. This makes it straightforward to generate documentation for an entire project.

30.4.2 Writing Output to Files

For larger projects, you'll want to write documentation to files rather than standard output. The `-o` (or `--output`) flag specifies an output directory:

Listing 30.18: Writing docs to a directory

```
// $ clat doc -o docs/ src/
// wrote docs/math_utils.md
// wrote docs/geometry.md
// wrote docs/string_helpers.md
```

Each input file produces a corresponding output file. The `.lat` extension is replaced with the appropriate extension for the chosen format (`.md` for Markdown, `.json` for JSON, `.html` for HTML, or `.txt` for plain text). The output directory is created automatically if it doesn't exist.

30.5 Output Formats: Markdown, JSON, HTML

The documentation generator supports four output formats, each suited to a different use case.

30.5.1 Markdown (Default)

Markdown is the default because it's the lingua franca of technical documentation—readable as plain text, renderable on GitHub, GitLab, and most wikis.

Listing 30.19: Generating Markdown output

```
// $ clat doc --md src/server.lat
// (or simply: clat doc src/server.lat)
```

The Markdown renderer produces headings, code-formatted signatures, tables for struct fields, and bulleted lists for enum variants.

30.5.2 Plain Text

For terminal viewing or piping to other tools:

Listing 30.20: Generating plain text output

```
// $ clat doc --text src/server.lat
```

The text format uses simple labels like `FUNCTIONS`, `STRUCTS`, and `ENUMS`, with dashed underlines and indented descriptions.

30.5.3 JSON

The JSON format is designed for machine consumption. It's what you'd use to build a custom documentation website, IDE plugin, or search index:

Listing 30.21: Generating JSON output

```
// $ clat doc --json src/math_utils.lat
```

The output is a JSON array where each element represents a source file. Each file object contains `"file"`, an optional `"module_doc"`, and an `"items"` array. Each item has a `"kind"` (`"function"`, `"struct"`, `"enum"`, `"trait"`, `"impl"`, or `"variable"`), a `"name"`, a `"line"` number, and optional `"doc"` text, plus kind-specific fields like `"params"` for functions or `"fields"` for structs.

Listing 30.22: Example JSON output (abbreviated)

```
// [
//   {
//     "file": "math_utils.lat",
//     "module_doc": "Utility functions for ...",
//     "items": [
//       {
//         "kind": "function",
//         "name": "clamp",
//         "line": 4,
//         "doc": "Clamp a value to the range ...",
//         "params": [
//           {"name": "value", "type": "Float"},
//           {"name": "low", "type": "Float"},
//           {"name": "high", "type": "Float"}
//         ],
//         "return_type": "Float"
//       }
//     ]
//   }
// ]
```

Building a Documentation Website

The JSON output pairs well with static site generators. Extract docs with `clat doc --json src/ > api.json`, then feed `api.json` into your favorite template engine to produce a searchable HTML site.

30.5.4 HTML

For a self-contained, styled documentation page:

Listing 30.23: Generating HTML documentation

```
// $ clat doc --html -o docs/ src/
```

The HTML renderer produces a complete web page with a dark theme, syntax-highlighted signatures, and interactive hover effects. Function signatures are color-coded—keywords in purple, types in gold, function names in blue—making it easy to scan a large API surface.

Struct fields render as tables, enum variants as styled lists, and trait methods as indented signatures under their parent trait.

30.5.5 The `-doc-format` Flag

As an alternative to the shorthand flags, you can use `--doc-format` with an explicit format name:

Listing 30.24: Using --doc-format

```
// $ clat doc --doc-format html src/
// $ clat doc --doc-format json src/
// $ clat doc --doc-format md src/
// $ clat doc --doc-format text src/
```

This is useful in scripts where you want to parameterize the format.

30.5.6 How Extraction Works Under the Hood

The documentation extractor in `src/doc_gen.c` uses its own lightweight scanner (`DocScanner`) rather than the full Lattice lexer. This is a deliberate design choice: the scanner needs to preserve doc comments while skipping implementation details, so it only recognizes enough syntax to identify declarations.

The process works as follows:

1. The scanner reads through the source file, skipping whitespace and regular comments.
2. When it encounters a `///` line, it reads consecutive doc comment lines into a single block, stripping the prefix and one optional leading space.
3. After the doc block, the scanner looks for a declaration keyword (`fn`, `struct`, `enum`, `trait`, `impl`, or a phase keyword).
4. For each declaration, the scanner extracts the name and kind-specific details: function parameters and return types, struct fields with their types, enum variants with optional tuple parameters, trait method signatures, and impl block methods.
5. The result is a `DocFile` structure (defined in `include/doc_gen.h`) containing a list of `DocItem` values that the renderers consume.

Items that lack doc comments are still extracted for functions, structs, enums, and traits. Variables, however, are only included if they have a doc comment, since most local or temporary variables aren't part of a module's public API.

Test Blocks Are Skipped

The doc generator recognizes `test "name" \{ ... \}` blocks and skips them entirely. Tests are implementation details, not part of your public documentation.

Exercises

1. Take one of the Lattice files you've written in earlier chapters and run `clat fmt` on it. Compare the before and after. Were there any formatting choices that surprised you? Try running with `--width 80` and `--width 120` to see how the output changes.
2. Create a small library module (perhaps a `temperature.lat` with conversion functions) and document every function, struct, and constant with `///` comments. Then run `clat doc --html -o docs/ temperature.lat` and open the resulting HTML in a browser.
3. Write a shell script (or Makefile target) that runs `clat fmt --check` on all `.lat` files in a directory tree and exits with a non-zero code if any file needs formatting. This is the skeleton of a CI formatting check.
4. Generate JSON documentation for a multi-file project and write a small Lattice script that reads the JSON and counts the total number of documented functions, structs, and enums across all files.

What's Next

With clean formatting and thorough documentation in hand, the natural next step is to make sure everything actually *works*. In Chapter 31, we'll explore Lattice's built-in testing system—inline `test` blocks that live right alongside your code, a rich set of assertion functions, and the `clat test` command that discovers and runs your entire test suite.

Chapter 31

Testing

Untested code is code that's waiting to betray you. You might trust it today—it ran once, it printed the right number—but refactor a function, add a parameter, swap a data structure, and suddenly the thing that "worked" doesn't. Lattice takes testing seriously: the language has built-in syntax for tests, a CLI runner that discovers and executes them, and a full suite of assertion functions that produce clear failure messages. No external framework to install, no configuration file to wrangle. Tests live right next to the code they verify.

31.1 `test "name" { body }` — Inline Tests

Lattice's `test` keyword lets you write tests directly in your source files. The syntax is:

Listing 31.1: Basic test block syntax

```
test "addition works" {
    assert(2 + 2 == 4, "basic arithmetic")
}
```

Test Block

A *test block* is a top-level declaration of the form `test "name" \{ body \}`. The name is a string literal that describes what the test verifies. The body contains any number of statements—typically assertions—that run when the test suite is invoked via `clat test`. Test blocks are *completely ignored* during normal program execution (`clat run`).

Let's write a more realistic example. Suppose we have a temperature conversion module:

Listing 31.2: Temperature conversion with inline tests

```
fn celsius_to_fahrenheit(c: Float) -> Float {
    return c * 9.0 / 5.0 + 32.0
}

fn fahrenheit_to_celsius(f: Float) -> Float {
    return (f - 32.0) * 5.0 / 9.0
}

test "freezing point conversion" {
    assert_eq(celsius_to_fahrenheit(0.0), 32.0)
    assert_eq(fahrenheit_to_celsius(32.0), 0.0)
}

test "boiling point conversion" {
    assert_eq(celsius_to_fahrenheit(100.0), 212.0)
    assert_eq(fahrenheit_to_celsius(212.0), 100.0)
}

test "body temperature" {
    let body_temp_f = celsius_to_fahrenheit(37.0)
    // 37.0 C should be approximately 98.6 F
    assert(body_temp_f > 98.5, "too low")
    assert(body_temp_f < 98.7, "too high")
}
```

The tests sit right below the functions they exercise. Anyone reading this file immediately understands what the functions are expected to do.

31.1.1 Tests Are Top-Level Declarations

Test blocks are parsed as top-level items by the Lattice parser (handled in `src/parser.c` as `ITEM_TEST`). They cannot appear inside functions, loops, or other blocks. This keeps them at the module level, visible and discoverable.

Listing 31.3: Tests belong at the top level

```
fn compute(x: Int) -> Int {
    return x * x
}

// This is correct --- test at the top level:
test "compute squares" {
    assert_eq(compute(5), 25)
    assert_eq(compute(0), 0)
    assert_eq(compute(-3), 9)
}
```

31.1.2 Tests Can Use Everything in the File

When `clat test` runs, it first executes all non-test top-level code (imports, function definitions, struct declarations, variable bindings) before running any test block. This means tests have access to every function, struct, enum, and global defined in the file.

Listing 31.4: Tests have access to all module-level definitions

```
struct Color {
    r: Int,
    g: Int,
    b: Int
}

fn mix(c1: Color, c2: Color) -> Color {
    return Color {
        r: (c1.r + c2.r) / 2,
        g: (c1.g + c2.g) / 2,
        b: (c1.b + c2.b) / 2
    }
}

test "mixing colors" {
    let red = Color { r: 255, g: 0, b: 0 }
    let blue = Color { r: 0, g: 0, b: 255 }
    let purple = mix(red, blue)

    assert_eq(purple.r, 127)
    assert_eq(purple.g, 0)
    assert_eq(purple.b, 127)
}
```

Tests Run in Their Own Scope

Each test block runs in a fresh scope. Variables defined inside one test are not visible to other tests. This prevents accidental coupling between test cases—each test is independent.

31.1.3 What Happens During Normal Execution?

When you run a file with `clat run`, test blocks are skipped entirely. The compiler (in `src/stackcompiler.c`) treats `ITEM_TEST` items as no-ops during normal compilation. This means you can sprinkle tests throughout your source code without any runtime cost in production.

Listing 31.5: Tests are invisible during normal execution

```
fn greet(name: String) -> String {
    return "Hello, " + name + "!"
}

test "greeting includes the name" {
    assert_contains(greet("Lattice"), "Lattice")
}

// When running: clat run greeter.lat
// Only the code below executes:
print(greet("World"))
// Output: Hello, World!
```

31.2 `clat test` — Running the Test Suite

The `clat test` command discovers and runs tests. At its most basic:

Listing 31.6: Running tests in a single file

```
// $ clat test temperature.lat
//   PASS  freezing point conversion
//   PASS  boiling point conversion
//   PASS  body temperature
//
// Results: 3 passed, 0 failed (3 total)
```

31.2.1 Directory Discovery

When you pass a directory instead of a file, `clat test` recursively searches for test files. The naming convention for test files is:

- Files matching `test_*.lat` (e.g., `test_math.lat`, `test_parser.lat`)
- Files matching `*_test.lat` (e.g., `math_test.lat`, `parser_test.lat`)

Listing 31.7: Running all tests in a directory

```
// $ clat test tests/
//   PASS  addition
//   PASS  subtraction
//   PASS  multiplication
//   FAIL  division by zero -- assertion failed: expected error
//
// Results: 3 passed, 1 failed (4 total)
```

The discovery is recursive, so tests nested in subdirectories are found too:

Listing 31.8: Typical test directory layout

```
// tests/
//   test_math.lat
//   test_strings.lat
//   integration/
//     test_server.lat
//     test_database.lat
```

All four files would be discovered and executed.

31.2.2 Filtering Tests by Name

The `-f` or `--filter` flag runs only tests whose name contains the given substring:

Listing 31.9: Filtering tests by name

```
// $ clat test tests/ -f "arithmetic"
//   PASS  basic arithmetic
//   PASS  floating-point arithmetic
//
// Results: 2 passed, 0 failed, 5 skipped (7 total)
```

The filter applies a simple substring match (using C's `strstr`, as seen in the evaluator's `evaluator_run_tests()` function in `src/eval.c`). If no tests match the filter, the runner reports how many were skipped.

31.2.3 Verbose Mode

Add `-v` or `--verbose` for more detailed output, including timing and setup information:

Listing 31.10: Verbose test output

```
// $ clat test -v tests/test_math.lat
```

31.2.4 Summary Mode

The `--summary` flag produces a concise summary after the test run, particularly useful in CI environments where you want a quick overview.

31.2.5 Full CLI Reference

Here is the complete set of `clat test` flags:

Flag	Description
`-f, --filter <pattern>`	Only run tests whose name matches pattern
`-v, --verbose`	Verbose output
`--summary`	Show summary after run
`--gc-stress`	GC stress testing mode
`--no-regions`	Disable region-based memory
`--no-assertions`	Disable runtime assertions
`-h, --help`	Show help

Exit Codes

`clat test` exits with code 0 if all tests pass, and code 1 if any test fails. This makes it straightforward to integrate into CI pipelines and shell scripts.

31.3 Assert Functions

Lattice provides a rich set of built-in assertion functions. These are not library functions—they're built into the evaluator and the runtime (`src/eval.c`, `src/runtime.c`), so they're always available without any imports.

31.3.1 assert(condition, message?)

The fundamental assertion. If the condition is falsy, the test fails with the optional message:

Listing 31.11: Basic assert usage

```
test "assert basics" {
    assert(1 == 1, "one equals one")
    assert(true, "true is truthy")
    assert(42, "non-zero integers are truthy")
    assert("hello", "non-empty strings are truthy")
}
```

When `assert` fails, it throws an error that the test runner catches. The failure message includes your custom string, making it clear which assertion broke:

Listing 31.12: A failing assert

```
test "this will fail" {
    let balance = -50
    assert(balance >= 0, "balance must not be negative")
    // Output: FAIL  this will fail -- assertion failed: balance must not be negative
}
```

31.3.2 assert_eq(actual, expected)

The workhorse of testing. Checks that two values are equal, and if not, shows both values in the failure message:

Listing 31.13: assert_eq with clear failure messages

```
test "string operations" {
    let greeting = "hello" + " " + "world"
    assert_eq(greeting, "hello world")

    let trimmed = "  spaces  ".trim()
    assert_eq(trimmed, "spaces")
}

test "array operations" {
    let numbers = [1, 2, 3, 4, 5]
    assert_eq(len(numbers), 5)
    assert_eq(numbers[0], 1)
    assert_eq(numbers[4], 5)
}
```

If the assertion fails, you get a message like: `// FAIL  string operations -- assert_eq failed: expected "space`

This level of detail in failure messages is worth its weight in gold during debugging. You don't need to add print statements—the assertion tells you exactly what went wrong.

31.3.3 `assert_ne(actual, expected)`

The opposite of `assert_eq`. Fails if the two values are equal:

Listing 31.14: Asserting inequality

```
test "unique ID generation" {
    let id1 = generate_id()
    let id2 = generate_id()
    assert_ne(id1, id2)
}
```

31.3.4 `assert_true(value)` and `assert_false(value)`

Strict boolean checks. Unlike `assert`, which accepts any truthy value, `assert_true` expects exactly `true` and `assert_false` expects exactly `false`:

Listing 31.15: Strict boolean assertions

```
test "boolean checks" {
    let is_admin = check_role("admin")
    assert_true(is_admin)

    let is_expired = check_expiry("2099-12-31")
    assert_false(is_expired)
}
```

assert_true Is Strict

`assert_true(42)` will *fail* because `42` is not the boolean value `true`, even though it's truthy. If you want truthy checking, use `assert(42, "expected truthy")`.

31.3.5 `assert_type(value, type_name)`

Verifies that a value has the expected type. The type name is a string that matches the output of `typeof()`, such as `"Int"`, `"Float"`, `"String"`, `"Array"`, or the name of a struct:

Listing 31.16: Type assertions

```
struct Point {
    x: Float,
    y: Float
}

test "type checking" {
    assert_type(42, "Int")
    assert_type(3.14, "Float")
    assert_type("hello", "String")
    assert_type([1, 2, 3], "Array")
    assert_type(true, "Bool")

    let origin = Point { x: 0.0, y: 0.0 }
    assert_type(origin, "Point")
}
```

The `assert_type` function checks both the primitive type and, for structs, the struct name. So `assert_type(origin, "Point")` succeeds because the evaluator (in `src/eval.c`) compares the struct's name field against the expected string.

31.3.6 `assert_contains(haystack, needle)`

Checks that a string or array contains a given element:

Listing 31.17: Containment assertions

```
test "string containment" {
    let message = "the quick brown fox"
    assert_contains(message, "brown")
    assert_contains(message, "fox")
}

test "array containment" {
    let colors = ["red", "green", "blue"]
    assert_contains(colors, "green")
}
```

31.3.7 `assert_throws(closure)`

Verifies that a piece of code throws an error. Pass a closure that should fail:

Listing 31.18: Asserting that code throws an error

```
test "division by zero throws" {
    assert_throws(|_| {
        let result = 10 / 0
    })
}

test "index out of bounds throws" {
    assert_throws(|_| {
        let items = [1, 2, 3]
        let bad = items[99]
    })
}
```

The built-in `assert_throws` wraps the closure in a `try`/`catch` block. If no error is thrown, the assertion fails with the message "expected an error but none was thrown."

31.3.8 Summary of Built-in Assertions

Function	Description
`assert(cond, msg?)`	Fails if `cond` is falsy
`assert_eq(a, b)`	Fails if `a != b`, showing both values
`assert_ne(a, b)`	Fails if `a == b`
`assert_true(v)`	Fails if `v` is not exactly `true`
`assert_false(v)`	Fails if `v` is not exactly `false`
`assert_type(v, t)`	Fails if `typeof(v)` is not `t`
`assert_contains(h, n)`	Fails if string/array `h` doesn't contain `n`
`assert_throws(fn)`	Fails if the closure does not throw

31.4 The Test Standard Library Module

Beyond the built-in assertions, Lattice ships with a test library at `lib/test.lat` that provides a more structured, framework-style testing experience. This is useful for larger projects that need test suites organized into groups.

31.4.1 Importing the Test Library

Listing 31.19: Importing the test library

```
import "lib/test" as t
```

31.4.2 Organizing Tests with `describe` and `it`

The test library provides two organizational primitives. The `describe()` function creates a named test suite, and `it()` creates individual test cases within a suite.

Listing 31.20: Structured test suites

```
import "lib/test" as t

t.run([
    t.describe("String operations", |_| {
        return [
            t.it("concatenation", |_| {
                t.assert_eq("hello" + " " + "world", "hello world")
            }),
            t.it("length", |_| {
                t.assert_eq(len("lattice"), 7)
            }),
            t.it("uppercase", |_| {
                t.assert_eq("hello".upper(), "HELLO")
            })
        ]
    }),
    t.describe("Math operations", |_| {
        return [
            t.it("addition", |_| {
                t.assert_eq(2 + 2, 4)
                t.assert_eq(1.5 + 2.5, 4.0)
            }),
            t.it("division by zero", |_| {
                t.assert_throws(|_| { 1 / 0 })
            })
        ]
    })
])
```

Running this with `clat run` (not `clat test`—this is a program, not a test file) produces:

Listing 31.21: Structured test output

```
// Running tests...
//
//   String operations
//     checkmark concatenation
//     checkmark length
//     checkmark uppercase
//
//   Math operations
//     checkmark addition
//     checkmark division by zero
//
// 5 passed, 0 failed, 5 total
// Completed in 2ms
```

Built-in Tests vs. the Test Library

The built-in `test "name" \{ ... \}` blocks are ideal for unit tests that live alongside source code, run by `clat test`. The `lib/test` library is better for standalone test programs that need more structure—nested suites, timing, and a custom runner. Both approaches work well; use whichever fits your project.

31.4.3 Extended Assertions in the Test Library

The test library provides additional assertion functions beyond the built-ins:

Listing 31.22: Extended assertions from lib/test

```
import "lib/test" as t

t.run([
    t.describe("Comparisons", |_| {
        return [
            t.it("greater than / less than", |_| {
                t.assert_gt(10, 5)
                t.assert_lt(3, 7)
                t.assert_gte(5, 5)
                t.assert_lte(5, 5)
            }),
            t.it("floating-point proximity", |_| {
                let pi = 3.14159265
                t.assert_near(pi, 3.14, 0.01)
            }),
            t.it("nil checks", |_| {
                t.assert_nil(nil)
                t.assert_not_nil(42)
                t.assert_not_nil("hello")
            }),
            t.it("type checking", |_| {
                t.assert_type(42, "Int")
                t.assert_type("hi", "String")
            })
        ]
    })
])
```

Here's the full list of assertions provided by `lib/test.lat`:

Function	Description
`t.check(cond)`	Fail if condition is falsy
`t.assert_eq(actual, expected)`	Fail if values differ
`t.assert_neq(actual, expected)`	Fail if values are equal
`t.assert_gt(a, b)`	Fail if `a <= b`
`t.assert_lt(a, b)`	Fail if `a >= b`
`t.assert_gte(a, b)`	Fail if `a < b`
`t.assert_lte(a, b)`	Fail if `a > b`
`t.assert_near(actual, expected, epsilon)`	Fail if values differ by more than `epsilon`

Function	Description
`t.assert_contains(haystack, needle)`	Fail if haystack doesn't contain needle
`t.assert_throws(closure)`	Fail if the closure doesn't throw
`t.assert_type(value, type_name)`	Fail if the type doesn't match
`t.assert_nil(value)`	Fail if value is not `nil`
`t.assert_not_nil(value)`	Fail if value is `nil`
`t.assert_true(value)`	Fail if value is not `true`
`t.assert_false(value)`	Fail if value is not `false`

assert_near for Floating-Point Tests

Comparing floating-point numbers with `assert_eq` can lead to spurious failures due to rounding. Use `t.assert_near(actual, expected, epsilon)` instead, where `epsilon` is the acceptable margin of error. For most calculations, an epsilon of `0.0001` is a good starting point.

31.4.4 How the Test Library Works Internally

If you open `lib/test.lat`, you'll find that the library is written entirely in Lattice. There's no magic—it uses the built-in `assert()` function combined with `format()` and `repr()` to produce clear error messages, and `try`/`catch` to recover from assertion failures.

The `it()` function returns a `Map` with `"name"` and `"fn"` keys. The `describe()` function calls the builder closure to get an array of these Maps and wraps them in a suite Map. The `run()` function iterates through suites and test cases, invoking each closure inside a `try`/`catch` block. If the closure completes without error, the test passes. If an assertion triggers an error, the catch block captures it and reports the failure.

31.5 Testing Strategies for Lattice Programs

Having the tools is one thing. Using them effectively is another. Let's explore strategies that experienced Lattice developers rely on.

31.5.1 Unit Testing: One Function, One Test

The most common pattern: write a test block for each public function. Keep tests close to the code they exercise.

Listing 31.23: Unit tests alongside the code

```
fn is_palindrome(s: String) -> Bool {
    let reversed = s.chars().reverse().join("")
    return s == reversed
}

test "is_palindrome with palindromes" {
    assert_true(is_palindrome("racecar"))
    assert_true(is_palindrome("level"))
    assert_true(is_palindrome("a"))
    assert_true(is_palindrome(""))
}

test "is_palindrome with non-palindromes" {
    assert_false(is_palindrome("hello"))
    assert_false(is_palindrome("lattice"))
}
```

31.5.2 Testing Edge Cases

The most productive tests often check boundaries: empty inputs, zero values, negative numbers, maximum sizes, and `nil`.

Listing 31.24: Testing edge cases

```
fn safe_divide(a: Float, b: Float) -> Float {
    if b == 0.0 {
        return 0.0
    }
    return a / b
}

test "safe_divide normal cases" {
    assert_eq(safe_divide(10.0, 2.0), 5.0)
    assert_eq(safe_divide(7.0, 3.5), 2.0)
}

test "safe_divide edge cases" {
    assert_eq(safe_divide(0.0, 5.0), 0.0)
    assert_eq(safe_divide(10.0, 0.0), 0.0)
    assert_eq(safe_divide(0.0, 0.0), 0.0)
    assert_eq(safe_divide(-10.0, 2.0), -5.0)
}
```

31.5.3 Testing Error Paths

Use `assert_throws` (either the built-in or the library version) to verify that functions fail when they should:

Listing 31.25: Testing that errors are thrown correctly

```
fn parse_positive_int(s: String) -> Int {
    let n = parse_int(s)
    if is_error(n) {
        return error("not a valid integer: " + s)
    }
    if n <= 0 {
        return error("expected positive integer, got " + to_string(n))
    }
    return n
}

test "parse_positive_int rejects invalid input" {
    assert_throws(|_| { parse_positive_int("abc") })
    assert_throws(|_| { parse_positive_int("-5") })
    assert_throws(|_| { parse_positive_int("0") })
}

test "parse_positive_int accepts valid input" {
    assert_eq(parse_positive_int("42"), 42)
    assert_eq(parse_positive_int("1"), 1)
}
```

31.5.4 Testing Structs and Methods

When testing structs, create instances with known values and verify field access, method results, and trait implementations:

Listing 31.26: Testing structs

```
struct Rectangle {
    width: Float,
    height: Float
}

fn area(r: Rectangle) -> Float {
    return r.width * r.height
}

fn perimeter(r: Rectangle) -> Float {
    return 2.0 * (r.width + r.height)
}

fn is_square(r: Rectangle) -> Bool {
    return r.width == r.height
}

test "rectangle area" {
    let r = Rectangle { width: 5.0, height: 3.0 }
    assert_eq(area(r), 15.0)
}

test "rectangle perimeter" {
    let r = Rectangle { width: 5.0, height: 3.0 }
    assert_eq(perimeter(r), 16.0)
}

test "square detection" {
    let square = Rectangle { width: 4.0, height: 4.0 }
    let rect = Rectangle { width: 4.0, height: 6.0 }
    assert_true(is_square(square))
    assert_false(is_square(rect))
}
```

31.5.5 Organizing a Test Suite

For a project of any size, keep tests in a dedicated `tests/` directory. Each test file mirrors the module it tests:

Listing 31.27: Project layout with tests

```
// src/
//   math.lat
//   strings.lat
//   http.lat
// tests/
//   test_math.lat
//   test_strings.lat
//   test_http.lat
```

Then run the entire suite with:

Listing 31.28: Running the full test suite

```
// $ clat test tests/
```

31.5.6 The Test-First Workflow

A powerful practice: write the test *before* the implementation.

Listing 31.29: Writing the test first

```
// Step 1: Write the test
fn fibonacci(n: Int) -> Int {
    // TODO: implement
    return 0
}

test "fibonacci sequence" {
    assert_eq(fibonacci(0), 0)
    assert_eq(fibonacci(1), 1)
    assert_eq(fibonacci(2), 1)
    assert_eq(fibonacci(5), 5)
    assert_eq(fibonacci(10), 55)
}

// Step 2: Run it (it fails)
// $ clat test fib.lat
//   FAIL  fibonacci sequence -- assert_eq failed: expected 1, got 0

// Step 3: Implement until it passes
```

The failing test gives you a clear target. Once all assertions pass, you know the implementation is correct—at least for the cases you specified.

31.5.7 Testing with GC Stress Mode

The `--gc-stress` flag triggers garbage collection after every allocation. This is invaluable for catching memory bugs:

Listing 31.30: Running tests with GC stress

```
// $ clat test --gc-stress tests/
```

Under GC stress, any test that accidentally holds a dangling reference or forgets to root a value will fail quickly. It makes tests slower, so you wouldn't run this in every CI build—but running it periodically catches subtle bugs that normal execution misses.

31.5.8 Disabling Assertions for Performance Testing

The `--no-assertions` flag disables runtime assertions (specifically, the `debug_assert` built-in). This is useful when you want to benchmark code that includes debug assertions without removing them from the source:

Listing 31.31: Running without runtime assertions

```
// $ clat test --no-assertions benchmarks/
```

Note that `assert()` (the testing assertion) is not affected by this flag—only `debug_assert()` is suppressed.

Exercises

1. Write a `stack.lat` module that implements a stack using an array (with `push`, `pop`, `peek`, and `is_empty` functions). Include inline `test` blocks that verify each operation, including edge cases like popping from an empty stack.
2. Create a structured test suite using `lib/test` that tests a string utility library with at least three `describe` groups and five total test cases. Use `assert_eq`, `assert_contains`, and `assert_throws`.
3. Write a test file that uses `assert_near` to verify floating-point calculations (trigonometry, compound interest, or physics formulas). Experiment with different epsilon values to understand how precision affects test results.
4. Set up a small project with three source files in `src/` and corresponding test files in `tests/`. Run the entire test suite, then use `--filter` to run only the tests related to one module. Write a one-line CI command that formats and tests the whole project.
5. Write a test that verifies the *behavior* of the phase system. For example, verify that freezing a value with `freeze` prevents mutation by catching the resulting error with `assert_throws`.

What's Next

When a test fails and the assertion message isn't enough to pinpoint the problem, you need a more powerful tool: a debugger. In Chapter 32, we'll explore Lattice's built-in CLI debugger, learn how to set breakpoints, step through code line by line, inspect variables, and even connect VS Code for a full graphical debugging experience.

Chapter 32

Debugging

Tests tell you *that* something broke. The debugger tells you *why*. Lattice includes a built-in interactive debugger that lets you pause execution, inspect variables, step through code line by line, and evaluate expressions—all without leaving the terminal. For developers who prefer a graphical environment, the debugger also speaks the Debug Adapter Protocol (DAP), enabling first-class integration with VS Code and other editors. And for the trickiest bugs—the ones where a value changed "somewhere" but you don't know where—Lattice's value history functions let you track mutations over time.

32.1 The `--debug` Flag and the CLI Debugger

To start a program under the debugger, add the `--debug` flag:

Listing 32.1: Launching the debugger

```
// $ clat run --debug server.lat
// Lattice debugger attached.
// Type 'help' for commands.
//
// Stopped at line 1 in <script>
//    1 | let host = "localhost"
//
// (dbg)
```

The program pauses at the very first line. You're now in the *debug REPL*—an interactive prompt where you can inspect state, set breakpoints, and control execution.

The Debug REPL

The debug REPL is a command loop that activates whenever execution is paused—either at startup in step mode, at a breakpoint, or after a step command completes. The prompt `(dbg)` indicates that the program is paused and waiting for your instruction. All debugger commands are processed by the `debugger_check()` function in `src/debugger.c`.

32.1.1 The Help Command

Type `help` (or `h`) at the debug prompt to see all available commands:

Listing 32.2: Debugger help output

```
// (dbg) help
// Debugger commands:
//   s, step           Step into (execute one line, enter calls)
//   n, next           Step over (execute one line, skip calls)
//   o, out, finish    Step out (run until current function returns)
//   c, continue       Continue until next breakpoint
//   l, locals         Show local variables
//   bt, stack         Show call stack / backtrace
//   list              Show source context around current line
//   b <line|func>     Set breakpoint at line or function
//   b <N> if <expr>   Set conditional breakpoint
//   d <id>            Delete breakpoint by ID
//   p <var>           Print variable value
//   eval <expr>       Evaluate expression
//   watch <expr>      Add watch expression
//   unwatch <id>      Remove watch expression
//   info b            List breakpoints
//   info w            List watch expressions
//   q, quit           Exit debugger
//   h, help           Show this help
```

32.1.2 Initial Breakpoint with `--break`

If you know roughly where the bug is, you can jump straight to a specific line instead of stepping from the beginning:

Listing 32.3: Starting with a breakpoint

```
// $ clat run --break 42 server.lat
// Lattice debugger attached.
// Breakpoints: line 42
//
// Breakpoint 1 at line 42
//    42 | let response = handle_request(request)
//
// (dbg)
```

The `--break` flag implies `--debug`. The program runs at full speed until it hits line 42, then pauses.

32.1.3 Viewing Source Context

The `list` command shows the source code surrounding the current line, with an arrow pointing to where execution is paused:

Listing 32.4: Viewing source context

```
// (dbg) list
//    37 |     let method = request.get("method")
//    38 |     let path = request.get("path")
//    39 |
//    40 |     if method == "GET" {
//    41 |         return handle_get(path)
// -->42 |     let response = handle_request(request)
//    43 |     } else if method == "POST" {
//    44 |         return handle_post(path, body)
//    45 |     }
//    46 |
//    47 |     return error("unsupported method: " + method)
```

The debugger loads the source file at startup (via `debugger_load_source()` in `src/debugger.c`) so it can display context around the current execution point.

32.1.4 Inspecting Variables

The `p` (or `print`) command shows the value of a variable:

Listing 32.5: Printing a variable

```
// (dbg) p request
// request = {method: "GET", path: "/api/users", headers: {...}}
//
// (dbg) p method
// method = "GET"
```

The `locals` command (or `l`) shows *all* local variables in the current scope:

Listing 32.6: Viewing all local variables

```
// (dbg) locals
//   request = {method: "GET", path: "/api/users", ...}
//   method = "GET"
//   path = "/api/users"
```

Under the hood, the debugger reads local variable names from the chunk's `local_names` array and their values from the VM's stack slots. If a variable has no debug name (e.g., a compiler-generated temporary), it won't appear in the listing.

32.1.5 Evaluating Expressions

The `eval` (or `e`) command evaluates an arbitrary Lattice expression in the context of the paused program:

Listing 32.7: Evaluating expressions in the debugger

```
// (dbg) eval len(path)
// => 11
//
// (dbg) eval method == "GET"
// => true
//
// (dbg) eval path.split("/")
// => ["", "api", "users"]
```

This is one of the most powerful features of the debugger. You can call functions, access fields, perform arithmetic—anything you could write in a normal Lattice expression. The evaluator compiles the expression on the fly and executes it on the live VM (see `debugger_eval_expr()` in `src/debugger.c`).

Side Effects in Eval

Expressions evaluated with `eval` run on the real VM state. If you evaluate something with side effects (e.g., `eval items.push(99)`), those effects will persist when you continue execution. Use `eval` for inspection, not mutation—unless you know what you're doing.

32.1.6 The Call Stack

The `bt` (or `stack` or `backtrace`) command shows the complete call chain:

Listing 32.8: Viewing the call stack

```
// (dbg) bt
// Backtrace:
//   #0 [line 12] in <script>
//   #1 [line 25] in process_request()
//   #2 [line 42] in handle_request()  <-- current
```

Frame `#0` is the bottom of the stack (the top-level script), and the frame marked `<-- current` is where execution is paused. This helps you understand how you got to the current line—especially useful when the same function is called from multiple places.

32.2 Breakpoints: Line, Function, Conditional

Breakpoints tell the debugger "stop here." Lattice supports three kinds: line breakpoints, function breakpoints, and conditional breakpoints.

32.2.1 Line Breakpoints

Set a breakpoint at a specific line number:

Listing 32.9: Setting a line breakpoint

```
// (dbg) b 15
// Breakpoint 1 set at line 15
//
// (dbg) c
// ... program runs ...
// Breakpoint 1 at line 15
//    15 | let total = calculate_total(items)
//
// (dbg)
```

The debugger assigns each breakpoint an auto-incrementing ID (starting at 1). When execution hits the breakpoint, the debugger shows the breakpoint number and line.

32.2.2 Function Breakpoints

If you don't know the line number but know the function name, set a function breakpoint:

Listing 32.10: Setting a function breakpoint

```
// (dbg) b calculate_total
// Breakpoint 2 set on function calculate_total()
//
// (dbg) c
// ... program runs ...
// Breakpoint 2 at calculate_total(), line 28
//    28 | fn calculate_total(items: Array) -> Float {
//
// (dbg)
```

Function breakpoints trigger whenever the named function is entered, regardless of which line called it. The debugger detects function entry by comparing the current frame count against the previous check—if the frame count increased and the new frame's function name matches, the breakpoint fires (see the function breakpoint check in `debugger_check()`).

32.2.3 Conditional Breakpoints

A conditional breakpoint only pauses when a Lattice expression evaluates to a truthy value. The syntax is `b <linefunc> if <expression>`:

Listing 32.11: Setting conditional breakpoints

```
// (dbg) b 15 if total > 1000.0
// Breakpoint 3 set at line 15 if total > 1000.0
//
// (dbg) b validate if input == ""
// Breakpoint 4 set on function validate() if input == ""
```

The condition is evaluated every time the breakpoint location is reached. If the condition is falsy, execution continues without pausing. If the condition throws an error, the debugger warns you and breaks anyway (so you can fix the condition).

Use Conditional Breakpoints for Loops

If you're debugging a loop that runs thousands of times but only the 500th iteration is wrong, a conditional breakpoint is far more effective than stepping through all 500 iterations:

```
// (dbg) b 20 if index == 500
```

32.2.4 Managing Breakpoints

List all breakpoints with `info b` (or `info breakpoints`):

Listing 32.12: Listing breakpoints

```
// (dbg) info b
// Num  Type      Enb  What
// 1    line      y    line 15  (hits: 3)
// 2    function  y    calculate_total()  (hits: 1)
// 3    line      y    line 15  if total > 1000.0  (hits: 0)
```

The table shows each breakpoint's ID, type (line or function), whether it's enabled (`y`/`n`), what it's set on, any condition, and how many times it has been hit.

Remove a breakpoint by its ID:

Listing 32.13: Deleting a breakpoint

```
// (dbg) d 2
// Breakpoint 2 removed
```

32.3 Step-Into, Step-Over, Step-Out

The three stepping commands are the bread and butter of interactive debugging. They control how much code executes before the debugger pauses again.

32.3.1 Step-Into (s)

The `s` (or `step`) command executes one line. If that line contains a function call, the debugger enters the function and pauses at its first line:

Listing 32.14: A simple program to step through

```
fn add_tax(price: Float, rate: Float) -> Float {
    let tax = price * rate
    return price + tax
}

let total = add_tax(100.0, 0.08)
print(total)
```

Listing 32.15: Stepping into a function call

```
// Stopped at line 6 in <script>
//     6 | let total = add_tax(100.0, 0.08)
//
// (dbg) s
// Stopped at line 2 in add_tax()
//     2 |     let tax = price * rate
//
// (dbg) p price
// price = 100.0
//
// (dbg) s
// Stopped at line 3 in add_tax()
//     3 |     return price + tax
//
// (dbg) p tax
// tax = 8.0
```

Step-into follows the actual execution path. If a line calls multiple functions, stepping enters the first one called.

32.3.2 Step-Over (`n`)

The `n` (or `next`) command executes one line but treats function calls as single steps—it doesn't enter them:

Listing 32.16: Stepping over a function call

```
// Stopped at line 6 in <script>
//     6 | let total = add_tax(100.0, 0.08)
//
// (dbg) n
// Stopped at line 7 in <script>
//     7 | print(total)
//
// (dbg) p total
// total = 108.0
```

The function `add_tax` executed fully, but the debugger didn't pause inside it. Step-over is the right choice when you trust the called function and want to see its result.

The implementation tracks the call depth: when `next` is issued, the debugger records the current frame count and only pauses when the frame count is at or below that level *and* the line has changed (see the `next_mode` and `next_depth` fields in the `Debugger` struct).

32.3.3 Step-Out (o)

The `o` (or `out` or `finish`) command runs until the current function returns, then pauses in the caller:

Listing 32.17: Stepping out of a function

```
// Stopped at line 2 in add_tax()
//     2 |     let tax = price * rate
//
// (dbg) o
// Stopped at line 7 in <script>
//     7 | print(total)
//
// (dbg) p total
// total = 108.0
```

Step-out is perfect when you've accidentally stepped into a function you don't care about. Rather than stepping through every line, `o` runs the rest of the function at full speed and returns you to the caller.

32.3.4 Continue (c)

The `c` (or `continue`) command resumes execution at full speed until the next breakpoint is hit (or the program ends):

Listing 32.18: Continuing execution

```
// (dbg) b 25
// Breakpoint 1 set at line 25
//
// (dbg) c
// ... program runs at full speed ...
// Breakpoint 1 at line 25
//    25 | let result = process(data)
//
// (dbg)
```

Choosing the Right Step Command

- **Step-into (s)**: Use when you want to see what happens *inside* a function call.
- **Step-over (n)**: Use when you trust the called function and want to move to the next line.
- **Step-out (o)**: Use when you're inside a function you don't care about and want to return to the caller.
- **Continue (c)**: Use when you've set breakpoints and want to skip ahead.

32.4 Watch Expressions and `breakpoint()`

32.4.1 Watch Expressions

A watch expression is monitored automatically every time the debugger pauses. Instead of typing `p total` at every stop, add a watch:

Listing 32.19: Adding watch expressions

```
// (dbg) watch total
// Watch 1: total
//
// (dbg) watch len(items)
// Watch 2: len(items)
```

Now every time the debugger pauses, it prints the current value of each watch expression:

Listing 32.20: Watch expressions displayed at each stop

```
// (dbg) s
// Stopped at line 18 in process()
//    18 |     total = total + item.price
//
// Watch expressions:
//   [1] total = 342.50
//   [2] len(items) = 15
//
// (dbg)
```

Watch expressions can be any valid Lattice expression—not just variable names. You can watch `items[0].name`, `total / len(items)`, or even a function call.

Manage watches with:

Listing 32.21: Managing watch expressions

```
// (dbg) info w
//   [1] total = 342.50
//   [2] len(items) = 15
//
// (dbg) unwatch 2
// Watch 2 removed
```

32.4.2 The `breakpoint()` Built-in

Sometimes the condition for pausing is too complex to express in a breakpoint condition string. For these cases, Lattice provides the `breakpoint()` built-in function. Call it anywhere in your code, and the debugger pauses at that point if the program is running under `--debug`:

Listing 32.22: Using breakpoint() in code

```
fn process_order(order: Map) -> Map {
    let items = order.get("items")
    let total = 0.0

    for item in items {
        total = total + item.get("price")

        // Pause if we encounter a suspiciously high price
        if item.get("price") > 10000.0 {
            breakpoint()
        }
    }

    return order
}
```

When the debugger is not attached (normal `clat run`), `breakpoint()` is a no-op. This means you can leave `breakpoint()` calls in your code without affecting production behavior.

Programmatic Breakpoints

`breakpoint()` is especially useful for catching rare conditions that are hard to reproduce. Rather than guessing which line to break on, embed the breakpoint directly in the condition logic:

```
if is_error(result) {
    breakpoint()  // pause only when something went wrong
}
```

32.5 DAP and VS Code Integration

The CLI debugger is powerful, but sometimes you want a graphical view: variable panes, inline source highlighting, and click-to-set breakpoints. Lattice supports the Debug Adapter Protocol (DAP), the industry standard used by VS Code, Neovim (via nvim-dap), Emacs (via dap-mode), and other editors.

32.5.1 What Is DAP?

The Debug Adapter Protocol, developed by Microsoft for VS Code, defines a JSON-based message protocol between an editor (the "client") and a debugger (the "adapter"). Instead of every editor implementing custom support for every language's debugger, DAP provides a common interface. Lattice's DAP implementation lives in `src/dap.c` and speaks DAP over stdin/stdout using Content-Length framed JSON messages.

32.5.2 Starting DAP Mode

To launch the debugger in DAP mode:

Listing 32.23: Launching DAP mode

```
// $ clat run --dap server.lat
```

In DAP mode, the debugger communicates via the Content-Length framed JSON protocol on stdin/stdout instead of the interactive REPL. The handshake follows the DAP specification: the client sends `initialize`, `launch`, optionally `setBreakpoints` and `setFunctionBreakpoints`, and finally `configurationDone`.

32.5.3 VS Code Configuration

To debug Lattice programs in VS Code, create a `.vscode/launch.json` file in your project:

Listing 32.24: VS Code launch.json for Lattice debugging

```
// {
//     "version": "0.2.0",
//     "configurations": [
//         {
//             "name": "Debug Lattice",
//             "type": "lattice",
//             "request": "launch",
//             "program": "server.lat",
//             "stopOnEntry": true
//         }
//     ]
// }
```

The `"stopOnEntry": true` option tells the debugger to pause at the first line (equivalent to `--debug` in CLI mode). If you set it to `false`, the program runs until it hits a breakpoint.

32.5.4 DAP Capabilities

Lattice's DAP adapter advertises the following capabilities during the initialize handshake (see `dap_handshake()` in `src/dap.c`):

Capability	Description
Configuration Done	Supports the `configurationDone` request
Function Breakpoints	Supports breakpoints on function names
Conditional Breakpoints	Supports breakpoint conditions
Evaluate for Hovers	Evaluates expressions when you hover in the editor

32.5.5 Features in the VS Code Debug View

Once connected, you get the full VS Code debugging experience:

- **Variables pane**: Shows all local variables and their values, organized by scope (Locals and Globals). Compound types (arrays, structs, maps) can be expanded to inspect their contents.
- **Call stack**: Displays the call chain from the top-level script down to the current function.

- **Breakpoint gutter**: Click in the gutter to set or remove line breakpoints. Right-click for conditional breakpoints.
- **Step controls**: The toolbar provides step-into, step-over, step-out, continue, and pause buttons.
- **Debug console**: Type expressions in the debug console to evaluate them, just like the CLI `eval` command.
- **Output capture**: `print()` output from your program appears in the Debug Console via DAP output events.

Compound Variable Expansion

When the Variables pane shows a struct, array, or map, VS Code can expand it to show individual fields or elements. This works through DAP's variable reference system: each compound value gets a unique reference ID, and when you expand it, VS Code sends a `variables` request with that reference. The implementation in `src/dap.c` uses a per-stop variable reference table (cleared on each stop event to avoid stale data).

32.5.6 Using with Other Editors

Any editor that supports DAP can connect to Lattice's debugger. For Neovim with nvim-dap:

Listing 32.25: Neovim nvim-dap configuration

```
// In your Neovim config (Lua):
// local dap = require("dap")
// dap.adapters.lattice = {
//     type = "executable",
//     command = "clat",
//     args = { "run", "--dap" }
// }
// dap.configurations.lattice = {{
//     name = "Debug Lattice",
//     type = "lattice",
//     request = "launch",
//     program = "${file}",
//     stopOnEntry = true,
// }}
```

The DAP protocol is identical regardless of the editor. Any tool that implements the DAP client side will work with Lattice's adapter.

32.6 Value History: `track()`, `history()`, `rewind()`

Some bugs aren't about the current state—they're about how the state got there. A variable has the wrong value, but when did it change? What was it before? Lattice's value history functions let you answer these questions without setting breakpoints on every assignment.

32.6.1 `track(variable)` — Start Recording

The `track()` function tells the runtime to record every value change for a named variable:

Listing 32.26: Tracking a variable

```
flux counter = 0
track(counter)

counter = 10
counter = 20
counter = 30
```

After calling `track(counter)`, every subsequent assignment to `counter` is recorded as a snapshot. The snapshot includes the value, its phase, the source line number, and the enclosing function name.

track(variable)

`track(variable)` enables value history tracking for the specified variable. The argument is a variable name (not a string—the compiler automatically converts it to a string behind the scenes). Tracking begins with an initial snapshot of the variable's current value. Subsequent mutations are recorded as they happen.

32.6.2 `history(variable)` — View the Timeline

Once you're tracking a variable, call `history()` to retrieve its full timeline:

Listing 32.27: Viewing variable history

```
flux temperature = 20.0
track(temperature)

temperature = 22.5
temperature = 25.0
temperature = 18.3

let timeline = history(temperature)
for entry in timeline {
    print(entry)
}
// Output:
// {phase: "flux", value: 20.0, line: 2, fn: nil}
// {phase: "flux", value: 22.5, line: 4, fn: nil}
// {phase: "flux", value: 25.0, line: 5, fn: nil}
// {phase: "flux", value: 18.3, line: 6, fn: nil}
```

Each entry in the timeline is a `Map` with four keys:

Key	Description
`phase`	The phase of the variable at the time of recording (e.g., `"flux"`, `"fix"`, `"let"`)
`value`	A deep copy of the variable's value at that moment
`line`	The source line number where the assignment occurred
`fn`	The name of the enclosing function, or `nil` for top-level code

The `phases()` function is a synonym for `history()` that returns the same data. Use whichever name feels more natural.

32.6.3 `rewind(variable, n)` — Look Back in Time

The `rewind()` function retrieves the value of a variable from `n` steps ago:

Listing 32.28: Rewinding variable state

```
flux score = 0
track(score)

score = 10
score = 25
score = 50
score = 100

print(rewind(score, 0))  // Output: 100 (current)
print(rewind(score, 1))  // Output: 50  (one step back)
print(rewind(score, 2))  // Output: 25  (two steps back)
print(rewind(score, 3))  // Output: 10  (three steps back)
print(rewind(score, 4))  // Output: 0   (initial value)
```

The parameter `n` counts backward from the most recent snapshot. `rewind(score, 0)` returns the current value; `rewind(score, 1)` returns the previous value, and so on. If `n` exceeds the number of recorded snapshots, `rewind()` returns `nil`.

The returned value is a *deep copy*—modifying it won't affect the original variable or the history.

32.6.4 Practical Example: Debugging a Running Total

Let's use value history to find where a running total goes wrong:

Listing 32.29: Debugging with value history

```
fn process_transactions(transactions: Array) -> Float {
    flux total = 0.0
    track(total)

    for tx in transactions {
        let amount = tx.get("amount")
        if tx.get("type") == "credit" {
            total = total + amount
        } else {
            total = total - amount
        }
    }

    // Something's wrong with the total.
    // Let's inspect the history:
    let timeline = history(total)
    for entry in timeline {
        print(format("Line {}: total = {} ({})",
            entry.get("line"),
            entry.get("value"),
            entry.get("phase")))
    }

    return total
}

let transactions = [
    {"type": "credit", "amount": 100.0},
    {"type": "credit", "amount": 50.0},
    {"type": "debit", "amount": 30.0},
    {"type": "credit", "amount": -20.0}
]

let result = process_transactions(transactions)
print("Final total: " + to_string(result))
// The history output shows exactly when the negative credit
// caused the unexpected subtraction.
```

By scanning the timeline, you can see exactly which transaction caused the total to deviate from expectations—and on which line.

32.6.5 Combining Tracking with the Debugger

Value history and the interactive debugger complement each other. You can set a breakpoint, then inspect the history at that point:

Listing 32.30: Using track() with the CLI debugger

```
// $ clat run --debug analytics.lat
//
// (dbg) b 45
// Breakpoint 1 set at line 45
//
// (dbg) c
// Breakpoint 1 at line 45
//    45 |     return average
//
// (dbg) eval history(running_sum)
// => [{phase: "flux", value: 0, line: 12, fn: "compute_average"},
//     {phase: "flux", value: 42, line: 18, fn: "compute_average"},
//     {phase: "flux", value: 85, line: 18, fn: "compute_average"},
//     ...]
//
// (dbg) eval rewind(running_sum, 3)
// => 42
```

This is the closest thing to time-travel debugging: you can inspect what a variable's value was at any prior point in the program, without having to re-run it.

Memory Cost of Tracking

Each tracked assignment creates a deep copy of the variable's value. For large data structures in tight loops, this can consume significant memory. Use `track()` judiciously—enable it for the specific variables you need to debug, and remove it when you're done.

Exercises

1. Write a program that computes the factorial of a number using recursion. Run it under `--debug` and step through the recursive calls using `s`. Use `bt` at the deepest recursion level to see the full call stack. Then use `o` to step out one frame at a time.

2. Create a program with a loop that processes an array of numbers. Set a conditional breakpoint that only triggers when the current number is negative. Use `eval` to inspect the loop state when the breakpoint fires.
3. Add `track()` to a variable inside a sorting algorithm (e.g., insertion sort). After the sort completes, print the full `history()` and use `rewind()` to inspect the variable's value at three different points during execution.
4. Set up a VS Code `launch.json` for a Lattice project. Set breakpoints by clicking in the gutter, then step through the program using the toolbar controls. Expand a struct and an array in the Variables pane to see their fields.
5. Write a program that uses `breakpoint()` inside an error-handling path. Run it normally (without `--debug`) to verify that `breakpoint()` has no effect, then run it under the debugger to confirm it pauses at the right moment.
6. Combine `watch` expressions with `track()` to debug a program that accumulates values in a loop. Set two watch expressions (one for the running total, one for the loop counter) and step through five iterations. Then use `rewind()` to verify the total at each step.

What's Next

You now have a complete toolkit for developing Lattice programs: formatting (`clat fmt`), documentation (`clat doc`), testing (`clat test`), and debugging (`--debug` and DAP). In Part X, we'll go under the hood and explore the internals of the Lattice runtime itself. Chapter 33 starts with the three compilation backends—tree-walk interpreter, stack VM, and register VM—and explains how your Lattice code is transformed from source text into executable instructions.

Part X

Under the Hood

Chapter 33

The Three Backends

Every line of Lattice code you write takes the same journey through the front end: the *lexer* turns source text into tokens, the *parser* builds an abstract syntax tree (AST), and the optional *phase checker* validates mutability constraints. What happens *after* that point—how the AST actually gets executed—depends on which **backend** you choose.

Lattice ships with three of them:

1. A **tree-walk interpreter** that traverses the AST node by node.
2. A **stack-based bytecode VM** that compiles the AST to a stream of one-byte opcodes and executes them against a value stack.
3. A **register-based VM** that compiles to fixed-width 32-bit instructions and operates on virtual registers.

Why three? Because each backend occupies a different point in the trade-off space between implementation complexity, startup latency, and peak throughput. In this chapter we will open each one up, see how it works, and learn when to reach for it.

33.1 Tree-Walk Interpreter

33.1.1 How It Works

The tree-walk interpreter is the oldest backend in Lattice. It lives in `src/eval.c` and is centred on a single recursive function, `eval_expr_inner`, that pattern-matches on the AST node type and returns an `EvalResult`. Statements are handled by `eval_stmt`; blocks by `eval_block_stmts`.

Tree-Walk Interpretation

A *tree-walk interpreter* executes a program by recursively visiting each node of the abstract syntax tree produced by the parser. There is no compilation step and no intermediate bytecode—the AST *is* the program representation at runtime.

Consider this tiny program:

Listing 33.1: A value flowing through the tree-walker

```
let width = 5
let height = 3
let area = width * height
print(area) // 15
```

When the tree-walker encounters the expression `width * height`, it:

1. Recognises a `BINARY_MUL` expression node.
2. Recursively evaluates the left child (`width`), looking it up in the current `Env` and finding the integer `5`.
3. Recursively evaluates the right child (`height`), finding `3`.
4. Multiplies the two values and wraps the result in an `EvalResult`.

Each recursion pushes a native C call frame. That means deeply nested expressions or long chains of function calls consume real stack space—the `Evaluator` struct tracks call depth and caps it at `max_call_depth` to prevent a segfault.

33.1.2 Environment and Scoping

The tree-walker manages variables through an `Env` (environment) structure—a hash map of names to `LatValues` with a pointer to the enclosing scope. Entering a block pushes a new child `Env`; leaving it pops the environment, freeing any bindings that went out of scope.

Listing 33.2: Scoping in the tree-walker

```
let outer = "hello"
if true {
  let inner = "world"
  print("${outer} ${inner}") // hello world
}
// inner is no longer accessible here
```

The tree-walker also tracks a rich set of metadata that the bytecode VMs do not carry: struct definitions (`struct_defs`), enum declarations (`enum_defs`), trait and impl registries, phase reactions, bonds, seed contracts, pressure constraints, and a full dual-heap garbage collector with region support. This makes the tree-walker the most *feature-complete* backend—it supports every language feature, including some (like `--stats` memory profiling) that are exclusive to it.

33.1.3 When to Use It

Invoke the tree-walker with the `--tree-walk` flag:

Listing 33.3: Running with the tree-walk backend

```
// command line:
// clat --tree-walk my_program.lat
```

Reach for the tree-walker when you need:

- **Memory profiling.** The `--stats` flag reports freezes, thaws, deep clones, scope depth, region usage, and GC statistics—data that only the tree-walker collects.
- **Maximum language coverage.** Every Lattice feature is supported here first; the bytecode backends sometimes lag behind for newly added syntax.
- **Debugging unusual behaviour.** Because execution follows the AST directly, the tree-walker's error messages can reference the exact AST node that failed, which is sometimes more informative than a bytecode offset.

Tree-Walker Performance

The tree-walker is the slowest backend. Every expression evaluation involves pointer chasing through AST nodes, C-level recursion overhead, and hash-map lookups for every variable access. For compute-heavy programs, the bytecode VMs can be 5–20× faster.

33.2 Stack-Based Bytecode VM

The stack-based bytecode VM is Lattice's **default backend**. When you run `clat my_program.lat` without any flags, this is the engine that fires.

33.2.1 Compilation

Before execution, the stack compiler (`src/stackcompiler.c`) walks the AST and emits a flat stream of bytes into a `Chunk` structure (defined in `include/chunk.h`). A `Chunk` holds:

- `code` — a dynamic array of `uint8_t` values: the bytecode stream.
- `constants` — a pool of `LatValues` referenced by index from the bytecode.
- `lines` — a parallel array mapping each byte offset to a source line number (used for error messages and debugging).
- `local_names` — debug metadata mapping stack slots to variable names.
- `pic` — a polymorphic inline cache for method dispatch (more on this shortly).

The compiler maintains a `Compiler` struct that tracks local variables as stack slots, manages scope depth, resolves upvalues for closures, and handles break/continue jump patching. Each function body is compiled into its own child `Chunk`, stored as a closure constant in the parent chunk's constant pool.

Listing 33.4: A function compiled to stack bytecode

```
fn add(a: Int, b: Int) -> Int {
  return a + b
}
let result = add(10, 20)
print(result) // 30
```

This compiles roughly to:

```
; Top-level chunk:
```

```
OP_CLOSURE       0       ; Push closure for add()
OP_DEFINE_GLOBAL "add"   ; Bind it globally
OP_GET_GLOBAL    "add"   ; Push the closure
OP_CONSTANT      10      ; Push argument 1
OP_CONSTANT      20      ; Push argument 2
OP_CALL          2       ; Call with 2 arguments
OP_DEFINE_GLOBAL "result"
OP_GET_GLOBAL    "result"
OP_PRINT         1       ; Print 1 value

; Function chunk for add(a, b):
OP_GET_LOCAL     1       ; Push a (slot 1; slot 0 = function itself)
OP_GET_LOCAL     2       ; Push b
OP_ADD                   ; Pop both, push sum
OP_RETURN                ; Return TOS
```

33.2.2 Opcodes

The full opcode set is defined in `include/stackopcode.h`. Opcodes are single bytes (0–255), giving the stack VM a compact instruction stream. Here are the major families:

Table 33.1: Stack VM opcode families

Family	Examples	Purpose
Stack manipulation	`OP_CONSTANT`, `OP_POP`, `OP_DUP`	Load values, discard or duplicate the top of stack
Arithmetic/logic	`OP_ADD`, `OP_MUL`, `OP_NOT`	Pop operands, push result
Comparison	`OP_EQ`, `OP_LT`, `OP_GTEQ`	Pop two values, push boolean
Variables	`OP_GET_LOCAL`, `OP_SET_GLOBAL`	Read/write locals (by slot) and globals (by name)
Control flow	`OP_JUMP`, `OP_LOOP`, `OP_JUMP_IF_FALSE`	Conditional and unconditional jumps
Functions	`OP_CALL`, `OP_CLOSURE`, `OP_RETURN`	Call, create closures, return
Data structures	`OP_BUILD_ARRAY`, `OP_BUILD_STRUCT`	Construct compound values from stack elements
Phase system	`OP_FREEZE`, `OP_THAW`, `OP_CLONE`	Phase transitions and deep cloning
Optimised paths	`OP_ADD_INT`, `OP_INC_LOCAL`, `OP_LOAD_INT8`	Fast paths for integer arithmetic

Some opcodes use one-byte operands (e.g. `OP_GET_LOCAL` takes a slot index); some use two-byte big-endian operands for larger ranges (`OP_JUMP`, `OP_LOOP`). When the constant pool exceeds 255 entries, wide variants like `OP_CONSTANT_16` kick in with 16-bit indices.

Exploring Opcodes

For a complete listing of every opcode with its encoding and semantics, see **??**. You can also call the C function `opcode_name()` (from `include/stackopcode.h`) to get the human-readable name of any opcode byte.

33.2.3 The Value Stack

At the heart of the stack VM is a fixed-size array of 4,096 `LatValue` slots:

```
LatValue stack[STACKVM_STACK_MAX];   // 4096 slots
LatValue *stack_top;                  // points past the last used slot
```

Every computation goes through this stack. To add two numbers, the VM pushes both onto the stack, then the `OP_ADD` handler pops them, adds them, and pushes the result. Local variables live directly in the stack—each call frame records a `slots` pointer marking the base of its window into the value stack.

Listing 33.5: Visualising the value stack

```
fn multiply(x: Int, y: Int) -> Int {
  let product = x * y
  return product
}
let answer = multiply(6, 7)
print(answer) // 42
```

When `multiply` is executing, the stack looks roughly like this:

```
slot 0: <multiply closure>     <-- frame->slots
slot 1: 6                      (x)
slot 2: 7                      (y)
slot 3: 42                     (product)
                               <-- stack_top
```

33.2.4 Call Frames

The VM maintains up to 512 call frames (`STACKVM_FRAMES_MAX`), each a `StackCallFrame` containing:

- `chunk` — the `Chunk` being executed in this frame.
- `ip` — the instruction pointer, pointing into the chunk's bytecode.
- `slots` — the base of this frame's region on the value stack.
- `upvalues` — pointers to closed-over variables for closures.

When a function returns via `OP_RETURN`, the VM pops the frame, restores the caller's instruction pointer, and pushes the return value onto the caller's stack.

33.2.5 The Dispatch Loop

The workhorse of the stack VM is the dispatch loop in `src/stackvm.c`—function `stackvm_run`. It reads one opcode at a time and branches to the handler for that opcode.

On compilers that support it (`GCC`, `Clang`), the dispatch loop uses **computed gotos**: a table of label addresses indexed by opcode value. After each handler finishes, it jumps directly to the next handler through the table, avoiding the overhead of a `switch` statement's bounds check and indirect branch prediction penalty. This is defined under the `VM_USE_COMPUTED_GOTO` preprocessor guard.

Computed Gotos

Computed gotos (`gcc`'s "labels as values" extension) let the VM replace a central `switch` with a dispatch table of `void*` pointers. Each handler ends with `goto *dispatch_table[next_opcode]`, which modern CPUs can predict more accurately than a single indirect branch. The result is typically a 10–20% speed improvement on tight loops.

33.2.6 Exception Handling and Defer

The stack VM implements `try`/`catch` with an exception handler stack (`STACKVM_HANDLER_MAX` = 64 entries). When the compiler encounters a `try` block, it emits `OP_PUSH_EXCEPTION_HANDLER` with a jump offset to the catch body. If `OP_THROW` fires, the VM unwinds frames back to the handler, builds a structured error map (with `message`, `line`, and `stack` fields), and pushes it onto the value stack for the catch clause.

`defer` works similarly: `OP_DEFER_PUSH` records the defer body's bytecode offset and scope depth. When execution leaves the scope (via `OP_DEFER_RUN`, return, or exception unwinding), deferred blocks execute in LIFO order, sharing the parent frame's stack slots.

Listing 33.6: Exception handling and defer in action

```
fn read_config(path: String) -> String {
  let file = open(path)
  defer { file.close() }

  try {
    return file.read_all()
  } catch err {
    print("Failed to read config: ${err["message"]}")
    return "{}"
  }
}
```

33.2.7 Upvalues and Closures

When a closure captures a variable from an enclosing scope, the compiler creates an `ObjUpvalue`. While the captured variable is still on the stack (i.e. the enclosing function has not returned), the upvalue's `location` pointer points directly into the stack slot. When the enclosing scope ends, `OP_CLOSE_UPVALUE` copies the value into the upvalue's `closed` field and redirects `location` to point there. This "open → closed" transition lets closures survive their parent's return without dangling pointers.

Listing 33.7: Closures capture variables via upvalues

```
fn make_counter() -> fn() -> Int {
  flux count = 0
  return || {
    count = count + 1
    return count
  }
}

let counter = make_counter()
print(counter()) // 1
print(counter()) // 2
print(counter()) // 3
```

After `make_counter` returns, the local `count` is closed into the upvalue. Each call to the returned closure reads and writes through the same `ObjUpvalue.closed` field.

33.3 Register-Based VM

The register-based VM is Lattice's newest backend. Invoke it with the `--regvm` flag. Where the stack VM manipulates an implicit operand stack, the register VM names its operands explicitly in each instruction—more like a physical CPU.

33.3.1 Instruction Encoding

Every register VM instruction is a fixed-width **32-bit word**, defined as `typedef uint32_t RegInstr` in `include/regopcode.h`. The encoding packs an 8-bit opcode and up to three operand fields:

Table 33.2: Register VM instruction formats

Format	Layout (bits)	Use
ABC	`[op:8][A:8][B:8][C:8]`	Three-address ops: `R[A] = R[B] + R[C]`
ABx	`[op:8][A:8][Bx:16]`	Load constant: `R[A] = K[Bx]`
AsBx	`[op:8][A:8][sBx:16]`	Conditional jump: `if !R[A]: ip += sBx`
sBx	`[op:8][sBx:24]`	Unconditional jump

Encoding and decoding use bit-manipulation macros:

```
// Encode: ROP_ADD into register 3, from registers 1 and 2
REG_ENCODE_ABC(ROP_ADD, 3, 1, 2)
//  => 0x02010308  (C=2, B=1, A=3, op=ROP_ADD)

// Decode:
REG_GET_OP(instr)   // opcode byte
REG_GET_A(instr)    // destination register
REG_GET_B(instr)    // first source
REG_GET_C(instr)    // second source
```

Fixed-width instructions mean the VM can index into the code array by instruction number rather than scanning variable-length byte sequences. This gives better instruction-cache behaviour and eliminates the need for "wide" opcode variants.

33.3.2 Register Windows

Each call frame in the register VM gets its own **register window**—a contiguous slice of 256 `LatValue` slots carved from the global register stack:

```
LatValue reg_stack[REGVM_REG_MAX * REGVM_FRAMES_MAX];
```

```
// = 256 * 64 = 16,384 register slots
```

When a function is called, the VM advances `reg_stack_top` by `REGVM_REG_MAX` (256) and points the new frame's `regs` pointer there. Function arguments are placed into the callee's low registers by the caller; the return value is written back to a register in the *caller's* window designated by `caller_result_reg`.

Listing 33.8: Register allocation for a function call

```
fn distance(x1: Float, y1: Float, x2: Float, y2: Float) -> Float {
  let dx = x2 - x1
  let dy = y2 - y1
  return (dx * dx + dy * dy).sqrt()
}

let d = distance(0.0, 0.0, 3.0, 4.0)
print(d) // 5.0
```

Inside `distance`, the register compiler assigns:

```
R[0] = <distance closure>
R[1] = x1 (0.0)
R[2] = y1 (0.0)
R[3] = x2 (3.0)
R[4] = y2 (4.0)
R[5] = dx  (3.0, result of ROP_SUB R[5], R[3], R[1])
R[6] = dy  (4.0, result of ROP_SUB R[6], R[4], R[2])
R[7] = dx*dx (9.0)
R[8] = dy*dy (16.0)
R[9] = sum   (25.0)
```

Notice the three-address style: `ROP_SUB R[5], R[3], R[1]` computes the subtraction and writes the result directly to R[5] without touching a stack. Intermediate values stay in registers instead of being pushed and popped, reducing memory traffic.

33.3.3 The Register Compiler

The register compiler (`src/regcompiler.c`) maintains a `RegCompiler` struct that is structurally similar to the stack compiler, but with a critical addition: a **register allocator**. Two functions manage the register file:

- `alloc_reg()` returns the next available register and advances the `next_reg` pointer (also updating a high-water mark `max_reg` used for bounded initialisation).

- `free_reg()` releases a register when its value is no longer needed (temporary expression results).

Local variables are assigned a permanent register for their entire lifetime within a scope. Temporary values get registers that are freed as soon as the value is consumed. This means the compiler must decide *at compile time* where every value lives—a more complex task than the stack compiler, which just pushes and pops.

33.3.4 Inline Caches

Both VMs use **polymorphic inline caches** (PICs) to speed up method dispatch, but they are especially important for the register VM. The PIC system is defined in `include/inline_cache.h`.

When the VM encounters a method call like `arr.push(42)`, it must determine the receiver's type, hash the method name, and search a dispatch table. A PIC short-circuits this:

1. On the first call, the full lookup runs and the result is cached in a 4-entry `PICSlot` keyed by `(type_tag, method_hash)`.
2. On subsequent calls, the VM checks the cache first. A hit returns the `handler_id` directly—an integer that indexes into a jump table, skipping the hash and string comparison entirely.
3. If the cache has four entries and a new type appears (megamorphic site), the oldest entry is evicted in FIFO order.

Listing 33.9: A polymorphic call site

```
fn describe(items: Array) -> String {
  // items could contain mixed types
  flux parts = []
  for item in items {
    // item.to_string() is polymorphic: Int, Float, String, etc.
    parts.push(item.to_string())
  }
  return parts.join(", ")
}

print(describe([1, 2.5, "three"])) // 1, 2.5, three
```

The PIC table uses a direct-mapped scheme with 64 slots per chunk, indexed by `ip_offset & 0x3F`. Collisions between different call sites in the same chunk are benign—they simply cause cache misses that trigger the slow path.

Handler IDs

Handler IDs are small integers defined as `#define` constants in `include/inline_cache.h`: `PIC_ARRAY_PUSH = 2`, `PIC_STRING_SPLIT = 32`, and so on. The special value `PIC_NOT_BUILTIN = 255` tells the VM to fall through to struct, map, or impl-block lookup.

33.3.5 RegChunk: The Register VM's Code Container

Where the stack VM stores bytecode in a `Chunk` of `uint8_t`, the register VM uses a `RegChunk` containing `uint32_t` instructions:

- `code` — array of `RegInstr` (32-bit words).
- `constants` — the same constant-pool design as the stack VM's `Chunk`.
- `magic` — the four bytes `0x524C4154` ("RLAT"), used to distinguish register chunks from stack chunks during serialisation.
- `max_reg` — the high-water register count, so the VM only needs to initialise that many registers per frame instead of all 256.
- `pic` — a `PICTable` for inline caching, same as the stack VM's `Chunk`.

Constant deduplication runs at compile time: the compiler scans the existing pool for matching strings, integers, and floats before adding a new entry. This keeps the pool compact and reduces memory pressure.

33.4 Choosing a Backend

33.4.1 Command-Line Flags

Table 33.3: Backend selection flags

Flag	Backend	When to use
(none)	Stack VM	General-purpose (default)
`--tree-walk`	Tree-walk interpreter	Memory profiling, full feature coverage
`--regvm`	Register VM	Performance-sensitive code

Listing 33.10: Selecting a backend from the command line

```
// Stack VM (default):
// clat my_program.lat

// Tree-walk interpreter:
// clat --tree-walk my_program.lat

// Register VM:
// clat --regvm my_program.lat
```

33.4.2 Feature Support

Not every feature is available on every backend. Here is a summary:

Table 33.4: Feature availability by backend

Feature	Tree-Walk	Stack VM	Reg VM
Core language (variables, functions, closures)			
Structs, enums, traits, impls			
Phase system (freeze, thaw, clone)			
Concurrency (scope/spawn, channels)			
Try/catch, defer			
Interactive debugger (`--debug`)	—		—
DAP protocol (`--dap`)	—		—
Memory stats (`--stats`)		—	—
GC modes (`--gc`, `--gc-stress`)			—
Pre-compiled bytecode (`.latc`/`.rlat`)	—		

33.4.3 The Execution Pipeline

Regardless of the backend you choose, the front end is shared. The pipeline in `src/main.c` looks like this:

1. **Lex**: `lexer_tokenize()` produces a vector of tokens.
2. **Parse**: `parser_parse()` produces a `Program` (the AST).
3. **Phase check** (strict mode): `phase_check()` validates phase annotations.
4. **Match check**: `check_match_exhaustiveness()` warns about non-exhaustive matches.

5. **Execute**:
    - Tree-walk: `evaluator_run(ev, &prog)`
    - Stack VM: `stack_compile(&prog, ...)` then `stackvm_run(&vm, chunk, ...)`
    - Register VM: `reg_compile(&prog, ...)` then `regvm_run(&vm, chunk, ...)`

Pre-Compiled Bytecode

You can skip steps 1–4 entirely by pre-compiling to `.latc` or `.rlat` files. We cover this in Chapter 34.

33.5 Performance Characteristics

33.5.1 Where the Time Goes

The three backends differ fundamentally in how they spend CPU cycles:

Tree-walker: dominated by pointer chasing (AST node traversal), hash-map lookups for variable access, and C-level recursion overhead. Each expression evaluation requires multiple function calls and dynamic dispatch on node types.

Stack VM: dominated by the dispatch loop's opcode fetch-decode-execute cycle. With computed gotos, the branch predictor handles the dispatch well, but every operation still requires pushing to and popping from the value stack—each a memory write and read.

Register VM: dominated by instruction decoding (bit shifts and masks) and direct register access. Because operands are named in the instruction itself, there is no push/pop overhead. The fixed-width encoding also means the instruction pointer advances by a predictable amount, improving prefetch behaviour.

33.5.2 Microbenchmark: Fibonacci

A classic (if imperfect) benchmark is recursive Fibonacci:

Listing 33.11: Fibonacci benchmark

```
fn fib(n: Int) -> Int {
  if n <= 1 { return n }
  return fib(n - 1) + fib(n - 2)
}

print(fib(35))
```

On a typical machine, the relative performance looks roughly like this:

Table 33.5: Relative Fibonacci performance (lower is better)

Backend	Relative Time
Register VM	1.0×
Stack VM	1.2–1.5×
Tree-walk interpreter	5–20×

The register VM wins because it avoids the stack VM's push/pop traffic and because its three-address instructions express the computation more densely. The tree-walker falls far behind due to per-node recursion and hash-map variable lookups.

Benchmarks Are Not the Whole Story

Fibonacci is function-call heavy and computation heavy, which exaggerates the dispatch overhead difference. For I/O-bound programs (web servers, file processing), the backend difference may be negligible because the bottleneck is the operating system, not the VM.

33.5.3 Startup Time

The tree-walker has the lowest startup time: it runs immediately from the AST with no compilation step. The stack VM and register VM both pay a compilation cost before the first instruction executes. For small scripts (a few dozen lines), this compilation overhead can exceed the total execution time.

If startup latency matters—say, for a CLI tool that runs and exits quickly—consider pre-compiling to bytecode (see Chapter 34).

33.5.4 Memory Usage

Tree-walker: keeps the full AST in memory throughout execution, plus the environment chain (hash maps). Memory usage scales with program size and scope depth.

Stack VM: the AST is freed after compilation; only the bytecode chunks and the value stack remain. The fixed-size value stack (4,096 slots) puts a ceiling on per-frame memory, though deep recursion can exhaust it.

Register VM: uses a fixed register stack of 16,384 slots (256 registers × 64 frames). The 32-bit instructions use 4× more bytes per instruction than the stack VM's byte stream, but there are typically *fewer* instructions because three-address ops replace sequences of push/pop operations.

33.5.5 Which One Should You Choose?

For most users, the answer is: **use the default** (stack VM). It has the best balance of features, performance, and stability. It supports the interactive debugger, pre-compiled bytecode, garbage collection, and has been the most thoroughly tested backend since Lattice v0.2.

Choose `--regvm` when you have a compute-heavy workload and want maximum throughput. Choose `--tree-walk` when you need memory profiling (`--stats`) or you are debugging a language feature that is not yet implemented in the bytecode backends.

33.6 Exercises

1. Write a Lattice program that computes the sum of integers from 1 to 1,000,000 using a `for` loop. Run it with all three backends and compare the execution times using your operating system's time utility (e.g. `time clat --regvm sum.lat`).
2. Modify the Fibonacci benchmark to use memoisation (store previously computed values in a `Map`). How does memoisation affect the relative performance of the three backends? Why?
3. Using the `--tree-walk --stats` flags, profile a program that creates and freezes several structs. Examine the output to understand how many freezes, deep clones, and scope pushes/pops occurred.
4. Read the opcode listing in **??**. Pick any "optimised" stack VM opcode (like `OP_ADD_INT` or `OP_INC_LOCAL`) and explain why a specialised opcode for integers is faster than the generic `OP_ADD`.
5. (Challenge) The register VM's `RegChunk` stores a `max_reg` field. Why is this useful? What would happen if the VM always initialised all 256 registers per frame?

What's Next Now that we know how Lattice code gets compiled and executed, the natural next question is: can we *save* that compiled form to disk? In Chapter 34, we will explore the `.latc` and `.rlat` file formats, learn how to pre-compile programs for faster startup, and meet Lattice's self-hosted compiler.

Chapter 34

Bytecode Serialization

Every time you run `clat my_program.lat`, Lattice lexes, parses, and compiles your source code before executing it. For a small script that is perfectly fine—the compiler is fast. But for larger projects, or for deployment scenarios where you want instant startup, you can skip all of that by *pre-compiling* your program to a bytecode file and running the result directly.

In this chapter we explore the `.latc` and `.rlat` file formats, walk through the serialization process byte by byte, learn how to use the `clat compile` command, and meet the self-hosted compiler—a Lattice program that compiles Lattice into bytecode.

34.1 The .latc and .rlat File Formats

Lattice defines two binary file formats, one for each bytecode VM:

- `.latc` — compiled stack VM bytecode. Magic bytes: `LATC` (`0x4C415443`).
- `.rlat` — compiled register VM bytecode. Magic bytes: `RLAT` (`0x524C4154`).

Both formats share the same overall structure. The implementation lives in `src/latc.c` and the header in `include/latc.h`.

34.1.1 Header

Every bytecode file starts with an 8-byte header:

The deserializer checks the magic bytes first. If they do not match, it rejects the file immediately with a clear error message. The format version allows the serialization format to evolve without breaking older files—if the version does not match the current `LATC_FORMAT` or `RLATC_FORMAT` constant, deserialization fails.

Table 34.1: Bytecode file header

Offset	Size	Field	Description
0	4 bytes	Magic	`"LATC"` or `"RLAT"`
4	2 bytes	Format version	Currently `1` for `.latc`, `2` for `.rlat` (little-endian `uint16`)
6	2 bytes	Reserved	Set to 0; available for future use

Bytecode File Formats

A `.latc` file contains serialized stack VM bytecodes (variable-width `uint8_t` opcodes). A `.rlat` file contains serialized register VM bytecodes (fixed-width `uint32_t` instructions). Both formats embed the full constant pool, line number tables, debug information, and nested function chunks.

34.1.2 Chunk Serialization (Stack VM)

After the header, the body of a `.latc` file is a serialized `Chunk`. The serializer (`serialize_chunk` in `src/latc.c`) writes these sections in order:

1. **Bytecode stream**: A `uint32_t` count followed by that many raw bytes of opcodes.
2. **Line number table**: A `uint32_t` count followed by that many `uint32_t` line numbers (one per bytecode byte, parallel to the code array).
3. **Constant pool**: A `uint32_t` count followed by tagged constants. Each constant starts with a type tag byte:
   - `0` (Int): followed by a little-endian `int64_t` (8 bytes)
   - `1` (Float): followed by a little-endian `double` (8 bytes)
   - `2` (Bool): followed by a single byte (0 or 1)
   - `3` (String): followed by a `uint32_t` length, then that many raw bytes
   - `4` (Nil): no payload
   - `5` (Unit): no payload
   - `6` (Closure): a nested function—followed by `uint32_t` param count, a variadic flag byte, then a recursively serialized child chunk
4. **Local names**: A `uint32_t` count, then for each slot: a presence byte (0 or 1), and if present, a length-prefixed string.

5. **Chunk name**: A presence byte, then (if present) a length-prefixed string for the function name used in stack traces.

Listing 34.1: A program whose bytecode we will examine

```
fn greet(name: String) -> String {
  return "Hello, ${name}!"
}

print(greet("Lattice"))
// Output: Hello, Lattice!
```

When this program is compiled, the top-level chunk contains the bytecodes for the script body (defining and calling `greet`). The constant pool includes the string `"Hello, "`, the string `"!"`, the function name `"greet"`, and a `TAG_CLOSURE` entry that recursively contains the function body's chunk.

Recursive Chunk Embedding

Function bodies are stored as closure constants in their parent chunk's constant pool. When the serializer encounters a `TAG_CLOSURE`, it recursively writes the child chunk. The deserializer mirrors this: it recursively reads the child chunk and constructs a `VAL_CLOSURE` value in the parent's constant pool. This tree of chunks mirrors the lexical nesting of functions in your source code.

34.1.3 RegChunk Serialization (Register VM)

The `.rlat` format follows the same pattern but with a few differences:

- **Instructions** are serialized as `uint32_t` values (4 bytes each, little-endian) instead of single bytes.
- **Closure constants** include an additional `uint32_t` upvalue count (the register VM stores this in the `region_id` field of the closure value).
- **max_reg**: A `uint8_t` appended at the end of each chunk, recording the high-water register count. This lets the VM initialize only the registers that are actually used, rather than clearing all 256 slots per frame.

Listing 34.2: Compiling for the register VM

```
// From the command line:
// clat compile --regvm greet.lat
// Produces: greet.rlat
//
// Run it:
// clat --regvm greet.rlat
```

34.1.4 Endianness and Portability

All multi-byte values in both formats are stored in **little-endian** byte order. The serializer writes bytes explicitly (e.g. `v & 0xff`, `(v >> 8) & 0xff`) rather than relying on platform byte order, so `.latc` and `.rlat` files are portable across architectures. A bytecode file compiled on an ARM Mac can be loaded and executed on an x86 Linux machine.

Inspecting Bytecode Files

You can use a hex editor or `xxd` to inspect the raw bytes of a `.latc` file. Look for the `LATC` magic at the start, then the `uint32_t` code length to know where the bytecodes end and the line table begins.

34.2 Pre-Compiling for Faster Startup

34.2.1 The Compile Command

The `clat compile` subcommand takes a `.lat` source file and writes a bytecode file:

Listing 34.3: Using clat compile

```
// Compile for stack VM (default):
// clat compile my_program.lat
// Produces: my_program.latc

// Compile with custom output path:
// clat compile my_program.lat -o build/app.latc

// Compile for register VM:
// clat compile --regvm my_program.lat
// Produces: my_program.rlat
```

Under the hood, `clat compile` follows the standard pipeline—lex, parse, compile—but instead of executing the bytecode, it serializes the resulting chunk to a file. The implementation lives in `src/main.c`: the compile subcommand handler reads the source, tokenizes and parses it, compiles to a `Chunk` (or `RegChunk` with `--regvm`), and calls `chunk_save()` or `regchunk_save()` to write the binary.

34.2.2 Running Pre-Compiled Bytecode

Once you have a bytecode file, you can run it directly:

Listing 34.4: Running pre-compiled bytecode

```
// Run stack VM bytecode:
// clat my_program.latc

// Run register VM bytecode:
// clat --regvm my_program.rlat
```

When Lattice detects a `.latc` or `.rlat` extension, it skips lexing, parsing, and compilation entirely. It reads the file, deserializes the chunk, and begins execution immediately. For large programs or deployment scenarios, this can meaningfully reduce startup time.

Listing 34.5: A build script that pre-compiles a project

```
// build.lat   pre-compile all source files
import "fs" as fs

let source_files = fs.glob("src/**/*.lat")
for file in source_files {
  let output = file.replace(".lat", ".latc")
  print("Compiling ${file} -> ${output}")
  // Shell out to the compiler:
  let result = exec("clat", ["compile", file, "-o", output])
  if result.exit_code != 0 {
    print("Error compiling ${file}: ${result.stderr}")
  }
}
```

Version Compatibility

Pre-compiled bytecode files are tied to the specific Lattice version that produced them. If you upgrade Lattice and the format version changes (even by one), old bytecode files will be rejected with an "unsupported format version" error. Always recompile when you update the language.

34.2.3 When to Pre-Compile

Pre-compilation is most useful when:

- **Deployment**: You are shipping a Lattice program to a server or container and want the fastest possible startup. The bytecode file is all you need—no source code required.
- **CI/CD pipelines**: Pre-compile once, run many times in different environments.
- **Large programs**: For projects with thousands of lines, skipping the parse and compile steps saves noticeable time.
- **Protecting source**: Distributing `.latc` files keeps your source code private (though bytecodes are not encrypted or obfuscated).

For development, pre-compilation is usually unnecessary—the compiler is fast enough that the overhead is imperceptible for most programs.

34.3 The Self-Hosted Compiler

34.3.1 A Compiler Written in Lattice

One of Lattice's most ambitious components lives in `compiler/latc.lat`: a self-hosted compiler that reads `.lat` source code, compiles it to stack VM bytecodes, and writes a `.latc` file—all written in Lattice itself.

Listing 34.6: Running the self-hosted compiler

```
// Using the self-hosted compiler:
// clat compiler/latc.lat input.lat output.latc
```

The self-hosted compiler implements the full compilation pipeline: a lexer (using Lattice's built-in `tokenize()` function), a recursive-descent parser, a bytecode emitter, and a binary serializer. It mirrors the structure of the C compiler in `src/stackcompiler.c` but is written entirely in Lattice.

34.3.2 Structure of the Self-Hosted Compiler

The compiler is organized into twelve sections within a single file:

1. **Opcode constants**: All stack VM opcodes are defined as integer constants (`let OP_CONSTANT = 0`, `let OP_ADD = 8`, etc.), matching the `Opcode` enum in `include/stackopcode.h`.

2. **Token type constants**: Token types for the lexer.

3. **Compiler state**: Global mutable state for the compilation process—the current chunk's code array, constant pool, local variables, scope tracking, and break/continue patching.

4. **Emit helpers**: Functions like `emit_byte`, `emit_bytes`, `emit_jump`, and `emit_loop` that write bytecodes to the current chunk.

5. **Scope management**: Functions to push/pop scopes, resolve locals, and manage upvalue capture.

6. **Parser**: A recursive-descent parser implementing Lattice's precedence-climbing expression grammar.

7. **Statement compilation**: Handlers for `let`, `flux`, `fix`, `if`, `for`, `while`, `match`, functions, structs, enums, and more.

8. **Expression compilation**: Handlers for binary operations, calls, field access, index expressions, closures, and string interpolation.

9. **Contract checking**: Support for `ensure` postconditions and return type annotations.

10. **Program compilation**: The top-level loop that iterates over all tokens and compiles each top-level statement.

11. **Serialization**: Functions that implement the `.latc` binary format: `serialize_chunk`, `serialize_constant`, `write_u32_le`, `write_i64_le`, etc.

12. **Main entry point**: Reads the input file, tokenizes, compiles, serializes, and writes the output.

Listing 34.7: The compiler's main function

```
fn main() {
  let argv = args()
  if len(argv) < 3 {
    eprint("usage: clat compiler/latc.lat <input.lat> <output.latc>")
    exit(1)
  }

  let input_path = argv[1]
  let output_path = argv[2]

  let source = read_file(input_path)
  if source == nil {
    eprint("error: cannot read '" + input_path + "'")
    exit(1)
  }

  tokens = tokenize(source)
  pos = 0
  compile_program()

  let ch_snap = snapshot_chunk()
  let buf = serialize_latc(ch_snap)
  let ok = write_file_bytes(output_path, buf)
  if !ok {
    eprint("error: cannot write '" + output_path + "'")
    exit(1)
  }
}
```

Notice how the compiler uses Lattice's built-in `tokenize()` function to lex the source code—it does not need to implement its own lexer from scratch. We will explore `tokenize()` in detail in Chapter 36.

34.3.3 How It Produces .latc Files

The self-hosted compiler builds up the chunk data as Lattice arrays: an array of bytecode values, an array of line numbers, and an array of tagged constants. When compilation is complete, it takes a "snapshot" of the current chunk state and passes it to the serializer.

The serializer writes the exact same binary format as the C implementation in `src/latc.c`: the `LATC` magic, the format version, and the recursive chunk structure. The resulting `.latc` file is byte-for-byte compatible with files produced by the C compiler.

Listing 34.8: Serialization in the self-hosted compiler

```
fn serialize_chunk(ch: any) {
  // ch = [code_arr, lines_arr, constants_arr, local_names_arr, chunk_name]
  let ch_code = ch[0]
  let ch_lines = ch[1]
  let ch_constants = ch[2]
  let ch_local_names = ch[3]

  // Code
  let code_len = len(ch_code)
  write_u32_le(code_len)
  for i in 0..code_len {
    write_u8(ch_code[i])
  }

  // Lines (same count as code)
  let lines_len = len(ch_lines)
  write_u32_le(lines_len)
  for i in 0..lines_len {
    write_u32_le(ch_lines[i])
  }

  // Constants (tagged)
  let const_len = len(ch_constants)
  write_u32_le(const_len)
  for i in 0..const_len {
    serialize_constant(ch_constants[i])
  }
  // ... local names and chunk name follow
}
```

34.4 Bootstrapping

34.4.1 The Chicken-and-Egg Problem

A self-hosted compiler is a compiler for language L written in language L. This creates a delightful paradox: to run the compiler, you need a Lattice implementation, but to create a Lattice implementation, you need a compiler.

Bootstrapping

Bootstrapping is the process of compiling a self-hosted compiler using an existing implementation of the language. The existing implementation (the "bootstrap compiler") runs the self-hosted compiler's source code, producing a compiled artifact that can then compile itself.

Lattice solves this the same way most self-hosted languages do: with a *bootstrap chain*. The C-based compiler (`src/stackcompiler.c`) serves as the bootstrap implementation. Here is how the chain works:

1. The C-based Lattice implementation (`clat`) can compile and run any `.lat` file.
2. We run the self-hosted compiler (`compiler/latc.lat`) *using* `clat`:

   ```
   clat compiler/latc.lat input.lat output.latc
   ```

3. The self-hosted compiler reads `input.lat`, compiles it to bytecodes, and writes `output.latc`.
4. The resulting `output.latc` can be run by `clat` without any source code.

The interesting part: the self-hosted compiler can also compile *itself*:

Listing 34.9: Bootstrapping the self-hosted compiler

```
// Step 1: Use the C bootstrap to compile the self-hosted compiler:
// clat compiler/latc.lat compiler/latc.lat latc.latc

// Step 2: Now latc.latc is a pre-compiled version of the compiler.
// We can use it to compile other programs:
// clat latc.latc some_program.lat some_program.latc

// Step 3: We can even use it to recompile itself:
// clat latc.latc compiler/latc.lat latc_v2.latc
```

If the self-hosted compiler is correct, then `latc.latc` and `latc_v2.latc` should produce identical bytecodes—a useful correctness check known as a *bootstrap comparison*.

34.4.2 Why Self-Hosting Matters

Self-hosting is more than a party trick. It provides concrete benefits:

- **Dogfooding**: The compiler exercises a huge slice of the language: arrays, maps, closures, string manipulation, file I/O, control flow, and even the built-in `tokenize()` function. Bugs in any of these features will cause the compiler to fail, providing an excellent integration test.
- **Portability**: The self-hosted compiler is pure Lattice with no C dependencies. It runs anywhere the Lattice runtime runs, including in environments where you cannot compile C code.
- **Extensibility**: Adding new compilation features (optimizations, new opcodes) can be done in Lattice rather than C, with a shorter development loop.
- **Proving the language**: A language that can compile itself has demonstrated a certain level of completeness and expressiveness.

Current Limitations

The self-hosted compiler currently targets only the stack VM. Register VM compilation (`.rlat`) is only available through the C-based `clat compile --regvm` command. As the register VM matures, a self-hosted register compiler may follow.

34.4.3 The Bootstrap in Practice

For day-to-day development, most Lattice users will never need to think about bootstrapping. The C-based `clat` is the standard way to compile and run programs. The self-hosted compiler exists as:

- A comprehensive integration test for the language.
- An example of a non-trivial Lattice program (nearly 5,000 lines).
- A proof that Lattice is expressive enough to write a compiler.
- A starting point for anyone who wants to hack on the compilation pipeline in Lattice rather than C.

Exercises

1. Compile a Lattice program to `.latc` using `clat compile`. Inspect the first 8 bytes of the resulting file with a hex editor or `xxd`. Verify that you see the `LATC` magic and the format version.
2. Write a program with at least three functions. Compile it to both `.latc` and `.rlat`. Compare the file sizes. Why is the `.rlat` file larger? (Hint: think about instruction width.)
3. Use the self-hosted compiler to compile a program:

```
clat compiler/latc.lat my_program.lat my_program.latc
```

Then run the result with `clat my_program.latc`. Verify the output matches running the source directly.

4. (Challenge) Perform a bootstrap comparison: use `clat` to compile the self-hosted compiler into `latc_v1.latc`. Then use `latc_v1.latc` to compile the self-hosted compiler again into `latc_v2.latc`. Compare the two files. Are they identical? What would it mean if they were not?

5. The serialization format stores line numbers for every bytecode byte—even for multi-byte instructions where only the first byte really needs a line number. Estimate how much space this wastes as a percentage of total file size for a typical program. Could a run-length encoding scheme reduce this?

What's Next We have seen how Lattice compiles and serializes its bytecodes. But what if you need to extend the language with capabilities written in C—image processing, database access, or hardware integration? In Chapter 35, we explore Lattice's native extension API: how to build shared libraries that plug into the runtime, the value constructors and accessors available to extension authors, and the collection of extensions that ship with Lattice.

Chapter 35

Native Extensions

Lattice's standard library covers a broad range of tasks—files, networking, JSON, cryptography. But eventually you will hit a wall: a C library for image processing, a database driver, or a hardware interface that does not exist in pure Lattice. Native extensions bridge that gap. They let you write performance-critical or platform-specific code in C, compile it into a shared library, and call it seamlessly from Lattice.

In this chapter we will learn the extension API, build an extension from scratch, survey the extensions that ship with Lattice, and understand how the runtime discovers and loads shared libraries.

35.1 The Extension API

35.1.1 The Contract

Every native extension is a shared library (`.dylib` on macOS, `.so` on Linux, `.dll` on Windows) that exports exactly one symbol: `lat_ext_init`. When Lattice loads your extension, it calls this function, passing an opaque `LatExtContext`. Your job is to call `lat_ext_register` to register each function you want to expose:

Extension Initialization

Every Lattice extension must export a function with the signature `void lat_ext_init(LatExtContext *ctx)`. Inside this function, call `lat_ext_register(ctx, name, fn)` once for each function you want to make available to Lattice code.

The entire public API lives in a single header: `include/lattice_ext.h`. Extensions compile against this header only—the internal `LatValue` layout is hidden behind opaque `LatExtValue` pointers, so your extension code never depends on Lattice internals.

35.1.2 Loading an Extension from Lattice

From Lattice code, you load an extension with `require_ext()`:

Listing 35.1: Loading a native extension

```
let db = require_ext("sqlite")

let conn = db["open"]("my_database.db")
let rows = db["query"](conn, "SELECT name, age FROM users WHERE age > ?", [21])

for row in rows {
  print("${row["name"]} is ${row["age"]} years old")
}

db["close"](conn)
```

The call `require_ext("sqlite")` returns a `Map` where each key is a function name (like `"open"`, `"query"`, `"close"`) and each value is a callable native closure. You call these functions exactly like any other Lattice function.

Map-Based Namespacing

Extensions return a `Map` rather than injecting globals. This keeps the global namespace clean and makes it explicit which extension a function came from. If you prefer dot-syntax, you can assign the map to a variable and use bracket notation: `db["query"](conn, sql)`.

35.1.3 The LatExtFn Signature

Every registered extension function has the same C signature:

```
typedef LatExtValue *(*LatExtFn)(LatExtValue **args, size_t argc);
```

The function receives an array of `LatExtValue` pointers (one per argument) and the argument count. It returns a heap-allocated `LatExtValue`, or `NULL` to return `nil`.

Here is a minimal extension function:

```
static LatExtValue *my_add(LatExtValue **args, size_t argc) {
```

```
    if (argc < 2) return lat_ext_error("add() expects 2 arguments");
    int64_t a = lat_ext_as_int(args[0]);
    int64_t b = lat_ext_as_int(args[1]);
    return lat_ext_int(a + b);
}
```

35.2 Value Constructors and Accessors

The extension API provides a complete set of constructors and accessors for working with Lattice values from C. All values are wrapped in an opaque `LatExtValue` pointer—you never touch raw `LatValue` structs.

35.2.1 Constructors

Table 35.1: Extension value constructors

Function	Returns	Description
`lat_ext_int(v)`	`LatExtValue*`	Create an integer value
`lat_ext_float(v)`	`LatExtValue*`	Create a float value
`lat_ext_bool(v)`	`LatExtValue*`	Create a boolean value
`lat_ext_string(s)`	`LatExtValue*`	Create a string (copies the input)
`lat_ext_nil()`	`LatExtValue*`	Create a nil value
`lat_ext_array(elems, len)`	`LatExtValue*`	Create an array from an element array
`lat_ext_map_new()`	`LatExtValue*`	Create an empty map
`lat_ext_error(msg)`	`LatExtValue*`	Create an error value (triggers a Lattice error)

All constructors return a heap-allocated `LatExtValue*`. The Lattice runtime deep-clones the value when receiving it, so you do not need to worry about the original being freed.

Error Handling

Returning `lat_ext_error("message")` from an extension function causes Lattice to raise a runtime error. The error message is available in `try`/`catch` blocks just like any other error.

35.2.2 Type Queries

Before accessing a value's contents, check its type:

```
LatExtType lat_ext_type(const LatExtValue *v);
```

The `LatExtType` enum includes: `LAT_EXT_INT`, `LAT_EXT_FLOAT`, `LAT_EXT_BOOL`, `LAT_EXT_STRING`, `LAT_EXT_ARRAY`, `LAT_EXT_MAP`, `LAT_EXT_NIL`, and `LAT_EXT_OTHER` for types not directly representable (like closures or structs).

35.2.3 Accessors

Table 35.2: Extension value accessors

Function	Returns	Description
`lat_ext_as_int(v)`	`int64_t`	Extract integer value
`lat_ext_as_float(v)`	`double`	Extract float value
`lat_ext_as_bool(v)`	`bool`	Extract boolean value
`lat_ext_as_string(v)`	`const char*`	Extract string pointer (borrowed)
`lat_ext_array_len(v)`	`size_t`	Get array length
`lat_ext_array_get(v, i)`	`LatExtValue*`	Get array element (deep clone)
`lat_ext_map_get(v, key)`	`LatExtValue*`	Get map value by key
`lat_ext_map_set(m, k, v)`	`void`	Set a map entry

Memory Management

Values returned by `lat_ext_array_get` and `lat_ext_map_get` are heap-allocated deep clones. You must call `lat_ext_free(v)` on them when you are done. The string returned by `lat_ext_as_string` is borrowed—do *not* free it.

35.2.4 Cleanup

Always free extension values you create for intermediate use:

```
void lat_ext_free(LatExtValue *v);
```

The return value of your extension function is freed automatically by the runtime after it has been cloned into the VM's value space. But any intermediate values you create (e.g., when iterating over an array argument) must be freed by your code.

35.3 Building a .dylib/.so Extension

Let us build a complete extension from scratch. We will create a `"math_extra"` extension that provides a fast integer exponentiation function.

35.3.1 Step 1: Write the C Code

Create a file called `math_extra.c`:

```
#include "lattice_ext.h"

void lat_ext_init(LatExtContext *ctx);

static LatExtValue *math_pow(LatExtValue **args, size_t argc) {
    if (argc < 2)
        return lat_ext_error("pow() expects 2 arguments");
    if (lat_ext_type(args[0]) != LAT_EXT_INT ||
        lat_ext_type(args[1]) != LAT_EXT_INT)
        return lat_ext_error("pow() expects integer arguments");

    int64_t base = lat_ext_as_int(args[0]);
    int64_t exp  = lat_ext_as_int(args[1]);
    int64_t result = 1;

    for (int64_t i = 0; i < exp; i++) {
        result *= base;
    }
    return lat_ext_int(result);
}

static LatExtValue *math_is_prime(LatExtValue **args, size_t argc) {
    if (argc < 1)
        return lat_ext_error("is_prime() expects 1 argument");
    int64_t n = lat_ext_as_int(args[0]);
    if (n < 2) return lat_ext_bool(false);
    for (int64_t i = 2; i * i <= n; i++) {
        if (n % i == 0) return lat_ext_bool(false);
    }
    return lat_ext_bool(true);
}

void lat_ext_init(LatExtContext *ctx) {
    lat_ext_register(ctx, "pow",      math_pow);
    lat_ext_register(ctx, "is_prime", math_is_prime);
}
```

35.3.2 Step 2: Write a Makefile

The build process is straightforward—compile the C file into a shared library, linking against the Lattice extension header:

```
CC     = cc
CFLAGS = -std=c11 -Wall -Wextra -fPIC -I../../include

UNAME_S := $(shell uname -s)
ifeq ($(UNAME_S),Darwin)
    SHARED_EXT = .dylib
    SHARED_FLAGS = -dynamiclib -undefined dynamic_lookup
else
    SHARED_EXT = .so
    SHARED_FLAGS = -shared
endif

TARGET = math_extra$(SHARED_EXT)

all: $(TARGET)

$(TARGET): math_extra.c
 $(CC) $(CFLAGS) $(SHARED_FLAGS) -o $@ $<

install: $(TARGET)
 @mkdir -p $(HOME)/.lattice/ext
 cp $(TARGET) $(HOME)/.lattice/ext/

clean:
 rm -f $(TARGET)
```

Key details:

- `-fPIC` generates position-independent code required for shared libraries.
- `-I../../include` points to the Lattice include directory where `lattice_ext.h` lives.
- On macOS, `-dynamiclib -undefined dynamic_lookup` creates a `.dylib` that resolves Lattice symbols at load time (they are provided by the `clat` binary).
- On Linux, `-shared` creates a `.so`.

35.3.3 Step 3: Build and Install

Listing 35.2: Building and installing an extension

```
// From the extension directory:
// make
// make install   (copies to ~/.lattice/ext/)
```

35.3.4 Step 4: Use It from Lattice

Listing 35.3: Using the math_extra extension

```
let math = require_ext("math_extra")

print(math["pow"](2, 10))       // 1024
print(math["pow"](3, 5))        // 243
print(math["is_prime"](17))     // true
print(math["is_prime"](100))    // false
```

Returning Compound Values

Extensions can return arrays and maps, not just scalars. Use `lat_ext_map_new()` and `lat_ext_map_set()` to build structured results—database rows, image metadata, or configuration objects.

35.4 Available Extensions

Lattice ships with six extensions in the `extensions/` directory. Each is a self-contained C project with its own `Makefile`.

35.4.1 ffi — Foreign Function Interface

The FFI extension lets you call functions from arbitrary shared libraries at runtime—no C code required. It supports struct marshalling, callback trampolines, memory operations, and type signatures.

Listing 35.4: Using the FFI extension

```
let ffi = require_ext("ffi")

// Open the C math library
let libm = ffi["open"]("/usr/lib/libm.dylib")

// Define a function signature: takes double, returns double
let sqrt_fn = ffi["sym"](libm, "sqrt", "d:d")

// Call it
let result = ffi["call"](sqrt_fn, 144.0)
print(result) // 12.0

ffi["close"](libm)
```

Core functions include `open`, `close`, `sym`, `call`, `addr`, `errno`, and `strerror`. Struct marshalling is available via `struct_define`, `struct_alloc`, `struct_get`, `struct_set`, and `struct_to_map`. Memory operations include `alloc`, `free`, `read_i32`, `write_i32`, and friends.

35.4.2 sqlite — SQLite Database

A complete SQLite client with parameterized queries, transactions, and metadata:

Listing 35.5: Using the SQLite extension

```
let db = require_ext("sqlite")

let conn = db["open"]("app.db")
db["exec"](conn, "CREATE TABLE IF NOT EXISTS users (name TEXT, age INT)")
db["exec"](conn, "INSERT INTO users VALUES (?, ?)", ["Alice", 30])
db["exec"](conn, "INSERT INTO users VALUES (?, ?)", ["Bob", 25])

let rows = db["query"](conn, "SELECT * FROM users WHERE age > ?", [20])
for row in rows {
  print("${row["name"]}: ${row["age"]}")
}
// Alice: 30
// Bob: 25

db["close"](conn)
```

Functions: `open`, `close`, `query`, `exec`, `status`, `last_insert_rowid`.

35.4.3 pg — PostgreSQL

A PostgreSQL client built on `libpq`:

Listing 35.6: Using the PostgreSQL extension

```
let pg = require_ext("pg")

let conn = pg["connect"]("host=localhost dbname=myapp")
let rows = pg["query"](conn, "SELECT id, email FROM accounts LIMIT 10")

for row in rows {
  print("${row["id"]}: ${row["email"]}")
}

pg["close"](conn)
```

35.4.4 redis — Redis Client

A minimal Redis client that implements the RESP protocol over raw TCP sockets—no external dependencies:

Listing 35.7: Using the Redis extension

```
let redis = require_ext("redis")

let conn = redis["connect"]("127.0.0.1", 6379)
redis["set"](conn, "greeting", "Hello from Lattice!")
let val = redis["get"](conn, "greeting")
print(val) // Hello from Lattice!

redis["incr"](conn, "counter")
redis["incr"](conn, "counter")
print(redis["get"](conn, "counter")) // 2

redis["close"](conn)
```

Functions: `connect`, `close`, `command`, `get`, `set`, `del`, `exists`, `expire`, `keys`, `incr`, `lpush`, `lrange`, `publish`, `ping`.

35.4.5 image — Image Metadata and Operations

Reads image headers directly (PNG, JPEG, GIF, BMP, WebP) without external library dependencies. Uses macOS `sips` for resize, convert, and thumbnail operations:

Listing 35.8: Using the image extension

```
let img = require_ext("image")

let info = img["info"]("photo.jpg")
print("Format: ${info["format"]}")   // jpeg
print("Size: ${info["width"]}x${info["height"]}")

// Resize (macOS only, uses sips):
img["resize"]("photo.jpg", 800, 600, "photo_small.jpg")
```

35.4.6 websocket — WebSocket Client and Server

Implements RFC 6455 WebSocket framing with support for both client and server roles, ping/pong, fragmented messages, and timeouts:

Listing 35.9: Using the WebSocket extension

```
let ws = require_ext("websocket")

// Client connection
let conn = ws["connect"]("ws://echo.websocket.org")
ws["send"](conn, "Hello, WebSocket!")
let response = ws["recv"](conn)
print(response) // Hello, WebSocket!
ws["close"](conn)
```

Functions include `connect`, `connect_auto` (auto-reconnect), `listen`, `accept`, `send`, `recv`, `recv_timeout`, `send_binary`, `close`, `status`, `ping`, and `set_timeout`.

35.5 Extension Search Paths

When you call `require_ext("name")`, the runtime searches for the shared library in the following order:

1. `./extensions/name.dylib` (or `.so` or `.dll`) — the local project's extensions directory.
2. `./extensions/name/name.dylib` — a subdirectory within the local extensions directory (the default layout for extensions that ship with the repository).
3. `~/.lattice/ext/name.dylib` — the user's global extension directory.
4. `$LATTICE_EXT_PATH/name.dylib` — a custom path set via the `LATTICE_EXT_PATH` environment variable.

The first match wins. If no library is found, `require_ext` raises an error listing all the paths that were searched.

Listing 35.10: Extension search path behavior

```
// If you have an extension in your project:
//   ./extensions/sqlite/sqlite.dylib   <-- found at path #2

// If you installed globally:
//   ~/.lattice/ext/sqlite.dylib        <-- found at path #3

// If you set a custom path:
//   LATTICE_EXT_PATH=/opt/lattice/ext clat my_program.lat
//   /opt/lattice/ext/sqlite.dylib      <-- found at path #4
```

Global Installation

Each extension's `Makefile` includes a `make install` target that copies the built library to `~/.lattice/ext/`. This makes the extension available to all your Lattice projects without copying files:

```
cd extensions/sqlite && make && make install
```

35.5.1 Security Considerations

Native extensions execute arbitrary C code with full process permissions. The runtime validates that the extension name does not contain path separators (`/` or `\`) to prevent path traversal attacks, but it does not sandbox the loaded code.

Trust Your Extensions

Only load extensions from sources you trust. A malicious extension has the same access as the Lattice process itself—it can read files, make network connections, or modify memory.

35.5.2 WASM Limitation

When Lattice is compiled to WebAssembly (via Emscripten), native extensions are not available. The `require_ext` function returns an error explaining that dynamic loading is unsupported in the WASM environment.

Exercises

1. Build the `math_extra` extension described in Section 35.3. Load it from a Lattice program and verify that `pow(2, 20)` returns `1048576`.
2. Add a `gcd` function to the `math_extra` extension that computes the greatest common divisor of two integers using Euclid's algorithm. Register it in `lat_ext_init` and test it from Lattice.
3. Write an extension function that accepts a Lattice array of strings and returns them sorted alphabetically using C's `qsort`. Use `lat_ext_array_len`, `lat_ext_array_get`, and `lat_ext_as_string` to extract the strings, sort them, and return a new array via `lat_ext_array`.
4. The extension API uses opaque `LatExtValue` pointers to hide the internal `LatValue` layout. What are the advantages of this design over exposing the raw struct? What are the disadvantages (hint: think about performance)?
5. (Challenge) Write an extension that wraps a third-party C library of your choice (e.g., `zlib` for compression, `curl` for HTTP requests). Include proper error handling for all failure cases.

What's Next We have seen how to extend Lattice with C code. But Lattice also provides powerful tools for inspecting and manipulating programs from *within* the language itself. In Chapter 36, we explore runtime code introspection, the `tokenize()` function, struct reflection, and the dynamic features that make Lattice a surprisingly flexible metaprogramming platform.

Chapter 36

Metaprogramming and Reflection

Most of the time, a program manipulates data: numbers, strings, arrays, structs. But sometimes a program needs to manipulate *itself*—inspect its own structure, examine types at runtime, build code from strings, or interrogate a struct to discover what fields it carries. These capabilities fall under the umbrella of **metaprogramming** and **reflection**.

Lattice is not a "macro language" in the tradition of Lisp or Rust. It does not have compile-time code generation or hygienic macros. Instead, it offers a pragmatic set of *runtime* introspection tools: functions that let you peek inside values, tokenize source code, convert between structs and maps, and even evaluate strings as Lattice code on the fly.

These tools are powerful, and like any powerful tool, they come with trade-offs. Let's explore them.

36.1 Runtime Code Introspection and Dynamic Features

Lattice provides a family of built-in functions for examining values at runtime. These are not methods on objects—they are global functions available everywhere, defined in `src/eval.c` (for the tree-walk interpreter) and mirrored in the bytecode VM dispatch loops.

36.1.1 Type Queries

The most fundamental introspection is asking "what *is* this value?"

Listing 36.1: Querying types at runtime

```
print(typeof(42))           // Int
print(typeof(3.14))         // Float
print(typeof("hello"))      // String
print(typeof(true))         // Bool
print(typeof([1, 2, 3]))    // Array
print(typeof(nil))          // Nil
```

The `typeof` function returns a string naming the value's type. Under the hood, it calls `builtin_typeof_str()` from `src/builtins.c`, which is a straightforward `switch` on the value's internal type tag. The possible return values are: `"Int"`, `"Float"`, `"Bool"`, `"String"`, `"Array"`, `"Struct"`, `"Closure"`, `"Unit"`, `"Nil"`, `"Range"`, `"Map"`, `"Channel"`, `"Buffer"`, `"Tuple"`, `"Set"`, `"Iterator"`, `"Ref"`, and `"Enum"`.

typeof()

`typeof(val)` returns a `String` naming the runtime type of `val`. For structs, it returns `"Struct"` (not the struct's name—use `struct_name()` for that).

This makes `typeof` useful for writing generic functions that need to branch on type:

Listing 36.2: A generic serializer using typeof

```
fn serialize(val: any) -> String {
  let t = typeof(val)
  if t == "Int" || t == "Float" || t == "Bool" {
    return to_string(val)
  }
  if t == "String" {
    return "\"" + val + "\""
  }
  if t == "Array" {
    let parts = val.map(|item| serialize(item))
    return "[" + parts.join(", ") + "]"
  }
  if t == "Nil" {
    return "null"
  }
  return repr(val)
}

print(serialize([1, "hello", true]))
// Output: [1, "hello", true]
```

36.1.2 Phase Queries

Lattice's phase system distinguishes fluid (mutable) values from crystallized (frozen) ones. The `phase_of` function lets you check a value's phase at runtime:

Listing 36.3: Querying value phase

```
flux temperature = 72.0
print(phase_of(temperature))     // fluid

fix pi = 3.14159
print(phase_of(pi))              // crystal

let config = freeze(["debug", "verbose"])
print(phase_of(config))          // crystal
```

The return values are `"fluid"`, `"crystal"`, or `"unphased"` (for values that have not been explicitly assigned a phase). This is particularly useful when writing library code that must respect its callers' mutability constraints:

Listing 36.4: Respecting caller's phase constraints

```
fn safe_append(collection: any, item: any) -> any {
  if phase_of(collection) == "crystal" {
    // Don't mutate a frozen collection; return a new one
    let thawed = thaw(collection)
    thawed.push(item)
    return freeze(thawed)
  }
  collection.push(item)
  return collection
}
```

36.1.3 String Representations

Two functions convert values to strings, each with a different purpose:

- `to_string(val)` — produces a human-readable display string. Strings are returned without quotes, arrays are formatted with brackets, and special values like `nil` become `"nil"`.
- `repr(val)` — produces a representation string that could (in principle) be parsed back into the original value. Strings are quoted, struct fields are shown, and closures display their parameter lists.

Listing 36.5: to_string vs. repr

```
let greeting = "hello"
print(to_string(greeting))   // hello
print(repr(greeting))        // "hello"

let items = [1, 2, 3]
print(to_string(items))      // [1, 2, 3]
print(repr(items))           // [1, 2, 3]

print(to_string(nil))        // nil
print(repr(nil))             // nil
```

When to Use repr()

Use `repr()` when writing debug output or logging. It shows you the "programmer's view" of a value, including type information that `to_string()` strips away. For user-facing output, prefer `to_string()` or string interpolation.

36.1.4 Dynamic Evaluation with lat_eval()

The most powerful (and most dangerous) introspection tool is `lat_eval()`, which takes a string of Lattice source code, parses and evaluates it at runtime, and returns the result:

Listing 36.6: Dynamic code evaluation

```
let result = lat_eval("2 + 3 * 4")
print(result)  // 14

// Define a function dynamically
lat_eval("fn greet(name: String) { print(\"Hello, \" + name) }")
greet("World")  // Hello, World
```

Under the hood, `lat_eval()` in `src/eval.c` does exactly what the top-level interpreter does: it tokenizes the string, parses it into an AST, registers any function and struct declarations into the current evaluator's registries, and executes the statements. Bindings created inside `lat_eval()` persist in the caller's scope.

lat_eval() and Security

`lat_eval()` executes arbitrary code with full access to the runtime environment. Never pass untrusted input (user data, network data) to `lat_eval()`. An attacker could read files, make network connections, or execute system commands. In production code, prefer structured approaches (closures, maps, pattern matching) over `lat_eval()`.

36.1.5 Completeness Checking with is_complete()

A companion to `lat_eval()` is `is_complete()`, which checks whether a source string has balanced brackets and parentheses:

Listing 36.7: Checking source completeness

```
print(is_complete("1 + 2"))        // true
print(is_complete("fn foo() {"))   // false  unclosed brace
print(is_complete("if true { }"))  // true
```

This is used internally by the REPL to determine whether to wait for more input or evaluate the current line. It tokenizes the string, counts bracket depth, and returns `true` if the depth is zero or less.

36.2 tokenize() — Inspecting Tokens

The `tokenize()` built-in function exposes Lattice's own lexer to user code. Given a string of source code, it returns an array of `Token` structs, each with a `type` field (the token type name) and a `text` field (the token's text):

Listing 36.8: Tokenizing Lattice source code

```
let tokens = tokenize("let x = 42 + y")

for tok in tokens {
  print("${tok.type}: ${tok.text}")
}
// Output:
// LET: LET
// IDENT: x
// ASSIGN: ASSIGN
// INT_LIT: 42
// PLUS: PLUS
// IDENT: y
```

Each element in the returned array is a struct with two fields:

- `type` — a `String` naming the token type (e.g., `"IDENT"`, `"INT_LIT"`, `"STRING_LIT"`, `"PLUS"`, `"LBRACE"`, `"FN"`).
- `text` — a `String` containing the token's text. For identifiers and literals, this is the actual text from the source. For operators and keywords, this is the token type name repeated.

The trailing `EOF` token is stripped from the result, so you only see meaningful tokens.

36.2.1 Use Cases for tokenize()

The most prominent use of `tokenize()` is in the self-hosted compiler (see Section 34.3). The compiler written in Lattice (`compiler/latc.lat`) calls `tokenize()` on the input source file to get its token stream, then implements a recursive-descent parser over those tokens. This means the self-hosted compiler does not need to reimplement lexing—it reuses the C lexer via this single function call.

Beyond the self-hosted compiler, `tokenize()` enables:

Listing 36.9: Counting keywords in a source file

```
fn count_keywords(source: String) -> Int {
  let tokens = tokenize(source)
  let keywords = ["fn", "let", "flux", "fix", "if", "else",
                  "for", "while", "return", "match", "struct"]
  flux count = 0
  for tok in tokens {
    if tok.type == "IDENT" {
      for kw in keywords {
        if tok.text == kw {
          count = count + 1
        }
      }
    }
  }
  return count
}
```

Listing 36.10: A syntax highlighter skeleton

```
fn highlight(source: String) -> String {
  let tokens = tokenize(source)
  flux result = ""
  for tok in tokens {
    if tok.type == "FN" || tok.type == "LET" || tok.type == "IF" {
      result = result + "\x1b[35m" + tok.text + "\x1b[0m "
    } else if tok.type == "STRING_LIT" {
      result = result + "\x1b[32m\"" + tok.text + "\"\x1b[0m "
    } else if tok.type == "INT_LIT" || tok.type == "FLOAT_LIT" {
      result = result + "\x1b[34m" + tok.text + "\x1b[0m "
    } else {
      result = result + tok.text + " "
    }
  }
  return result
}
```

tokenize() and the Bytecode VMs

The `tokenize()` function is available in all three backends. In the tree-walk interpreter, it calls `lexer_tokenize()` directly. In the bytecode VMs, it is dispatched as a native built-in function that invokes the same C lexer. The returned tokens are identical regardless of backend.

36.2.2 Token Types

Lattice's lexer produces a rich set of token types. Here is a representative sample:

For a complete list, see **??**.

36.3 Struct Reflection Functions

Lattice structs are not opaque—at runtime, you can interrogate them to discover their name, fields, and values. This makes it possible to write generic code that operates on *any* struct, regardless of its definition.

Table 36.1: Common token types returned by tokenize()

Token Type	Example Text	Description
IDENT	my_var	Identifier
INT_LIT	42	Integer literal
FLOAT_LIT	3.14	Float literal
STRING_LIT	hello	String literal (without quotes)
FN	FN	fn keyword
LET	LET	let keyword
FLUX	FLUX	flux keyword
FIX	FIX	fix keyword
IF	IF	if keyword
PLUS	PLUS	+ operator
ASSIGN	ASSIGN	= assignment
LBRACE	LBRACE	{ opening brace
RBRACE	RBRACE	} closing brace

36.3.1 struct_name()

Listing 36.11: Getting a struct's type name

```
struct User {
  name: String,
  age: Int,
}

let alice = User { name: "Alice", age: 30 }
print(struct_name(alice))  // User
```

The `struct_name()` function returns the struct's type name as a string. This is distinct from `typeof(alice)`, which returns `"Struct"` for all struct instances. With `struct_name()` you can distinguish between different struct types at runtime.

36.3.2 struct_fields()

Listing 36.12: Listing a struct's field names

```
struct Point {
  x: Float,
  y: Float,
}

let origin = Point { x: 0.0, y: 0.0 }
let fields = struct_fields(origin)
print(fields)  // [x, y]
```

The `struct_fields()` function returns an array of field name strings. The names are in declaration order—the same order you specified when defining the struct.

36.3.3 struct_to_map()

Perhaps the most useful reflection function, `struct_to_map()` converts a struct instance into a `Map` where the keys are field names and the values are deep clones of the field values:

Listing 36.13: Converting a struct to a map

```
struct Config {
  host: String,
  port: Int,
  debug: Bool,
}

let cfg = Config { host: "localhost", port: 8080, debug: true }
let map = struct_to_map(cfg)

print(map["host"])    // localhost
print(map["port"])    // 8080
print(map["debug"])   // true
```

This is invaluable for serialization. Instead of writing a custom serializer for each struct type, you can convert to a map and pass it to `json_stringify()`:

Listing 36.14: Generic struct serialization via struct_to_map

```
fn to_json(val: any) -> String {
  if typeof(val) == "Struct" {
    let map = struct_to_map(val)
    return json_stringify(map)
  }
  return json_stringify(val)
}

struct Product {
  name: String,
  price: Float,
  in_stock: Bool,
}

let item = Product { name: "Widget", price: 9.99, in_stock: true }
print(to_json(item))
// Output: {"name":"Widget","price":9.99,"in_stock":true}
```

36.3.4 struct_from_map()

The inverse of `struct_to_map()` is `struct_from_map()`, which creates a struct instance from a type name and a map of field values:

Listing 36.15: Creating a struct from a map

```
struct User {
  name: String,
  age: Int,
}

let data = Map::new()
data["name"] = "Bob"
data["age"] = 25

let user = struct_from_map("User", data)
print(user.name)  // Bob
print(user.age)   // 25
```

The struct type must already be defined in the current scope. If a field is missing from the map, it defaults to `nil`. This function is particularly useful for deserialization—parsing JSON into a map, then converting the map into a typed struct:

Listing 36.16: JSON deserialization into a struct

```
struct Task {
  title: String,
  done: Bool,
  priority: Int,
}

fn parse_task(json_str: String) -> any {
  let map = json_parse(json_str)
  return struct_from_map("Task", map)
}

let task = parse_task("{\"title\": \"Write docs\", \"done\": false, \"priority\": 1}")
print(task.title)     // Write docs
print(task.priority)  // 1
```

36.3.5 Combining Reflection for Generic Programming

With these four functions, you can write powerful generic code:

Listing 36.17: A generic equality checker for structs

```
fn struct_equal(a: any, b: any) -> Bool {
  if typeof(a) != "Struct" || typeof(b) != "Struct" {
    return false
  }
  if struct_name(a) != struct_name(b) {
    return false
  }
  let fields_a = struct_fields(a)
  let fields_b = struct_fields(b)
  if len(fields_a) != len(fields_b) {
    return false
  }
  let map_a = struct_to_map(a)
  let map_b = struct_to_map(b)
  for field in fields_a {
    if map_a[field] != map_b[field] {
      return false
    }
  }
  return true
}

struct Point {
  x: Float,
  y: Float,
}

let p1 = Point { x: 1.0, y: 2.0 }
let p2 = Point { x: 1.0, y: 2.0 }
let p3 = Point { x: 3.0, y: 4.0 }

print(struct_equal(p1, p2))  // true
print(struct_equal(p1, p3))  // false
```

Listing 36.18: A generic struct printer

```
fn describe_struct(val: any) -> String {
  if typeof(val) != "Struct" {
    return repr(val)
  }
  let name = struct_name(val)
  let fields = struct_fields(val)
  let map = struct_to_map(val)
  flux parts = []
  for field in fields {
    parts.push(field + ": " + repr(map[field]))
  }
  return name + " { " + parts.join(", ") + " }"
}

struct Color {
  r: Int,
  g: Int,
  b: Int,
}

let coral = Color { r: 255, g: 127, b: 80 }
print(describe_struct(coral))
// Output: Color { r: 255, g: 127, b: 80 }
```

36.4 format() — String Building

While string interpolation with `${}` handles most formatting needs, the `format()` function provides a positional placeholder approach similar to Python's `str.format()` or Rust's `format!()` macro.

36.4.1 Basic Usage

Listing 36.19: Using format() for string building

```
let msg = format("{} is {} years old", "Alice", 30)
print(msg)  // Alice is 30 years old

let report = format("Total: {} items, {} in stock", 150, 42)
print(report)  // Total: 150 items, 42 in stock
```

The format string uses `{}` as a placeholder. Arguments are consumed left to right, one per placeholder. Each argument is converted to its display string (equivalent to calling `to_string()` on it).

36.4.2 Escaping Braces

If you need a literal `{` or `}` in the output, double it:

Listing 36.20: Escaping braces in format strings

```
let json_template = format("{{\"name\": \"{}\", \"age\": {}}}", "Bob", 25)
print(json_template)  // {"name": "Bob", "age": 25}
```

The implementation in `src/format_ops.c` handles this by checking for consecutive `{{` and `}}` patterns, emitting a single brace for each pair.

36.4.3 Error Handling

If you provide fewer arguments than placeholders, `format()` raises a runtime error:

Listing 36.21: format() error on missing arguments

```
// This will raise an error:
// let msg = format("{} and {}", "Alice")
// Error: format: not enough arguments for placeholders
```

Extra arguments are silently ignored—only the placeholders that exist in the format string consume arguments.

36.4.4 format() vs. String Interpolation

When should you use `format()` instead of string interpolation?

- **Dynamic format strings**: When the template is not known at compile time—for instance, loaded from a configuration file or received over a network.
- **Reusable templates**: When you want to define a template once and apply it with different arguments.
- **Array-based arguments**: When the arguments come from a computed source rather than inline variables.

Listing 36.22: Dynamic templates with format()

```
// A logging system with configurable format
flux log_template = "[{}] {}: {}"

fn log(level: String, message: String) {
  let timestamp = to_string(time_ms())
  print(format(log_template, timestamp, level, message))
}

log("INFO", "Server started")
// Output: [1709234567890] INFO: Server started

log("ERROR", "Connection refused")
// Output: [1709234567891] ERROR: Connection refused
```

Prefer Interpolation for Readability

For most cases, string interpolation (`"Hello, ${name}!"`) is more readable than `format("Hello, {}!", name)`. Reserve `format()` for situations where the template itself is dynamic or where you are building strings programmatically.

36.5 The Power and Danger of Runtime Features

Runtime introspection and dynamic evaluation are a double-edged sword. Let's be honest about both edges.

36.5.1 The Power

Dynamic features let you build things that would otherwise require code generation or preprocessors:

Listing 36.23: A configuration-driven plugin system

```
struct Plugin {
  name: String,
  version: String,
  init: fn() -> Bool,
}

fn load_plugin_from_config(config: Map) -> any {
  let name = config["name"]
  let source = read_file("plugins/" + name + ".lat")
  lat_eval(source)
  // The plugin registers itself via a global function
  return true
}
```

Listing 36.24: A test runner that discovers tests

```
fn find_test_functions(source: String) -> Array {
  let tokens = tokenize(source)
  flux test_names = []
  for i in 0..len(tokens) - 1 {
    if tokens[i].type == "TEST" {
      if i + 1 < len(tokens) && tokens[i + 1].type == "STRING_LIT" {
        test_names.push(tokens[i + 1].text)
      }
    }
  }
  return test_names
}

let source = read_file("my_tests.lat")
let tests = find_test_functions(source)
print("Found " + to_string(len(tests)) + " tests:")
for name in tests {
  print("  - " + name)
}
```

Listing 36.25: A generic ORM-style mapper

```
fn map_rows_to_structs(rows: Array, struct_type: String) -> Array {
  flux results = []
  for row in rows {
    let instance = struct_from_map(struct_type, row)
    results.push(instance)
  }
  return results
}

struct Employee {
  name: String,
  department: String,
  salary: Float,
}

// Imagine these rows came from a database query
let rows = [
  Map::new(),
  Map::new(),
]
rows[0]["name"] = "Alice"
rows[0]["department"] = "Engineering"
rows[0]["salary"] = 95000.0
rows[1]["name"] = "Bob"
rows[1]["department"] = "Marketing"
rows[1]["salary"] = 85000.0

let employees = map_rows_to_structs(rows, "Employee")
for emp in employees {
  print("${emp.name} in ${emp.department}")
}
// Output:
// Alice in Engineering
// Bob in Marketing
```

36.5.2 The Dangers

Every runtime feature trades away some combination of safety, performance, and readability.

Performance. `lat_eval()` must lex, parse, and interpret the string on every call. There is no caching and no compilation—it is a full interpreter invocation. In a tight loop, this overhead is devastating.

Security. `lat_eval()` on untrusted input is a code injection vulnerability. The evaluated code runs with the same privileges as the host program.

Debuggability. Dynamically generated code does not have source file locations. Error messages will reference the evaluated string, not a file and line number. Stack traces from `lat_eval()` are harder to read.

Type safety. Reflection functions like `struct_to_map()` erase type information. The resulting map has string keys and `any`-typed values—you lose the compiler's ability to check field names and types.

Refactoring. Code that uses `lat_eval()` to define functions or structs dynamically is invisible to static analysis tools, formatters, and documentation generators. Renaming a function in your editor will not catch references inside evaluated strings.

The lat_eval() Rule of Thumb

If you find yourself reaching for `lat_eval()`, stop and ask: "Can I solve this with closures, maps, or pattern matching instead?" The answer is almost always yes. Reserve `lat_eval()` for genuinely dynamic scenarios: plugin systems, REPLs, code generators, and development tools.

36.5.3 Guidelines for Responsible Metaprogramming

1. **Prefer closures over eval.** Instead of building a function as a string and evaluating it, accept a closure parameter and call it.

2. **Use struct reflection for serialization, not for business logic.** `struct_to_map()` and `struct_from_map()` are ideal for converting between data formats. They are less ideal for implementing core application behavior.

3. **Validate before evaluating.** If you must use `lat_eval()`, validate the input first. At minimum, check that it tokenizes without errors using `is_complete()`.

4. **Use tokenize() for analysis, not execution.** `tokenize()` is safe—it only reads, never evaluates. It is the right tool for syntax highlighting, code analysis, and documentation generation.

5. **Document dynamic behavior.** When your code uses reflection or dynamic evaluation, add comments explaining *why* the dynamic approach is necessary and what the expected inputs are.

Listing 36.26: Closures instead of eval: the better way

```
// BAD: building code as strings
// let code = "fn process(x: Int) -> Int { return x * " + to_string(multiplier) + " }"
// lat_eval(code)

// GOOD: use a closure that captures the value
fn make_processor(multiplier: Int) -> fn(Int) -> Int {
  return |x| x * multiplier
}

let process = make_processor(3)
print(process(10))  // 30
```

36.6 Exercises

1. Write a function `pretty_print(val: any)` that produces a formatted, indented string representation of any value. For structs, use `struct_name()` and `struct_to_map()` to show the struct type and fields. For arrays and maps, indent nested elements.
2. Use `tokenize()` to write a function that counts the number of functions defined in a Lattice source file. Look for the `FN` token followed by an `IDENT` token.
3. Write a `diff_structs(a: any, b: any)` function that compares two struct instances of the same type and returns an array of field names that differ. Use `struct_fields()` and `struct_to_map()` to implement it generically.
4. Create a simple calculator REPL that reads a line from the user with `input()`, evaluates it with `lat_eval()`, and prints the result. Add basic error handling: catch evaluation errors and display a helpful message instead of crashing.
5. (Challenge) Write a "struct validator" function that takes a struct instance and a map of `{field_name: validation_closure}` pairs. For each field, call the corresponding validation closure with the field's value and collect any failures. Return an array of error messages for fields that failed validation.

What's Next We have peered behind the curtain at Lattice's runtime introspection capabilities—type queries, tokenization, struct reflection, and dynamic evaluation. With the "Under the Hood" part complete, we are ready to put everything together. In Chapter 37, we will build a complete command-line tool from scratch, applying the language features, concurrency patterns, and standard library knowledge we have accumulated throughout this book.

Part XI

Real-World Lattice

Chapter 37

Building a CLI Tool

Every language earns its stripes when you build something real with it. So far we have explored Lattice's features one by one—phases, concurrency, pattern matching, modules. Now it is time to weave them together into a complete, polished command-line application: a file-search utility called `find-it` that searches files for a pattern and prints matching lines with context.

Along the way we will use three libraries from the Lattice ecosystem: `cli` for argument parsing, `dotenv` for loading configuration from `.env` files, and `log` for structured, leveled logging. By the end of this chapter you will have a template for every CLI tool you write in Lattice.

37.1 The `cli` Library for Argument Parsing

A command-line tool lives or dies by its interface. Users expect `--help` to work, short flags like `-v` to combine, and helpful error messages when they get something wrong. The `cli` library (found in `lib/cli.lat`) gives us all of this through a declarative API.

37.1.1 Creating a Parser

We begin by importing the library and creating a new application definition:

Listing 37.1: A minimal CLI parser

```
import "lib/cli" as cli

let app = cli.new("greet", "A friendly greeter")
app.flag("loud", "l", "Shout the greeting")
app.option("name", "n", "Who to greet", "World")

let opts = app.parse()
// opts is a Map with keys: "loud", "name", "_args"
```

The `cli.new` function takes two arguments—the program name and a short description—and returns a Map packed with closures for configuring and executing the parser.

CLI Application Map

The value returned by `cli.new` is a Map containing closures: `.flag(name, short, desc)` registers a boolean flag, `.option(name, short, desc, default)` registers a named option with a default value, `.arg(name, desc, required)` registers a positional argument, `.help()` generates help text, and `.parse()` parses the command-line arguments and returns a result Map.

37.1.2 Flags, Options, and Positional Arguments

The `cli` library distinguishes three kinds of command-line input:

- **Flags** are boolean switches. They default to `false` and become `true` when present. Think `--verbose` or `-v`.
- **Options** carry a value. They can appear as `--output file.txt`, `--output=file.txt`, or `-o file.txt`. Every option has a default.
- **Positional arguments** are the bare values that remain after flags and options are consumed. They can be required or optional.

Listing 37.2: Registering all three kinds of input

```
import "lib/cli" as cli

let app = cli.new("findi", "Search files for a pattern")

// Flags (boolean)
app.flag("verbose", "v", "Show extra information")
app.flag("count", "c", "Print only the count of matches")
app.flag("ignorecase", "i", "Case-insensitive search")

// Options (with default values)
app.option("output", "o", "Write results to a file", "")
app.option("context", "C", "Lines of context to show", "0")

// Positional arguments
app.arg("pattern", "The search pattern", true)     // required
app.arg("path", "File or directory to search", false) // optional
```

37.1.3 Parsing and the Result Map

Calling `app.parse()` reads the process arguments (via the built-in `args()` function), matches them against the registered flags, options, and positionals, and returns a result Map. Flags map to `Bool` values, options map to `String` values (or their defaults), and positional arguments map to `String` values. Any extra positional arguments land in the `"_args"` key as an array.

Listing 37.3: Working with parsed results

```
let opts = app.parse()

// Boolean flags
let is_verbose = opts.get("verbose")    // true or false

// String options (always strings---convert if needed)
let ctx_lines = to_int(opts.get("context"))

// Positional arguments
let pattern = opts.get("pattern")       // guaranteed present (required)
let path = opts.get("path")             // may be nil if not provided
```

Options Are Always Strings

The `cli` library returns all option values as strings, including numeric defaults. When you register `app.option("port", "p", "Port number", "8080")`, the parsed value will be the string `"8080"`, not the integer `8080`. Use `to_int()` or `to_float()` to convert.

37.1.4 Automatic Help and Error Handling

The parser automatically intercepts `--help` and `-h`, printing a nicely formatted help message and exiting. You never need to register a help flag yourself.

Running our `findi` tool with `--help` produces:

Listing 37.4: Auto-generated help output

```
findi - Search files for a pattern

Usage: findi [OPTIONS] <pattern> [path]

Arguments:
  <pattern>     The search pattern (required)
  [path]        File or directory to search

Options:
  -v, --verbose       Show extra information
  -c, --count         Print only the count of matches
  -i, --ignorecase    Case-insensitive search
  -o, --output        Write results to a file [default: ]
  -C, --context       Lines of context to show [default: 0]
  -h, --help          Show this help message
```

If a user provides an unknown flag or forgets a required argument, the parser prints an error along with the help text and exits with code 1.

37.1.5 Combined Short Flags

Like many Unix tools, the `cli` library supports combining short flags into a single token. The invocation `findi -vic "TODO" src/` is equivalent to `findi -v -i -c "TODO" src/`. When combining, every character except the last must be a flag. The last character may be an option, in which case the next argument becomes its value:

Listing 37.5: Combined flags with a trailing option

```
# -v and -i are flags; -C is an option consuming "3"
findi -viC 3 "pattern" src/
```

Double-Dash Separator

The `cli` library supports the `--` separator. Everything after `--` is treated as a positional argument, even if it starts with a dash. This is essential for patterns like `findi -- -TODO- file.txt` where the pattern itself looks like a flag.

37.2 The `dotenv` Library for Configuration

CLI tools often need configuration that varies between environments—database URLs, API keys, default directories. Hardcoding these into source files is fragile. The `dotenv` library (in `lib/dotenv.lat`) loads key-value pairs from `.env` files into the process environment, so your code can read them with `env()`.

37.2.1 The Basics: `load` and `env`

Suppose our search tool needs a default search directory. We create a `.env` file alongside our script:

Listing 37.6: A `.env` file

```
# Default configuration for findi
FINDI_DEFAULT_PATH=./src
FINDI_MAX_RESULTS=100
FINDI_LOG_LEVEL=info
```

Then we load it at startup:

Listing 37.7: Loading a .env file

```
import "lib/dotenv" as dotenv

dotenv.load()  // loads .env from the current directory

let default_path = env("FINDI_DEFAULT_PATH")  // "./src"
let max_results = to_int(env("FINDI_MAX_RESULTS"))  // 100
```

The `dotenv.load()` function is deliberately forgiving: if no `.env` file exists, it silently does nothing. This is ideal for production, where configuration typically comes from the real environment rather than a file.

37.2.2 Loading Specific Files

When you need to load from a particular path—say, a per-environment configuration—use `dotenv.load_file`:

Listing 37.8: Loading a specific .env file

```
import "lib/dotenv" as dotenv

dotenv.load_file("config/.env.production")
```

Unlike `load()`, this function will halt with an error if the file does not exist. It is meant for situations where the configuration file is mandatory.

37.2.3 Advanced Loading with Options

For more control, `dotenv.load_opts` accepts a Map of options:

Listing 37.9: Loading with options

```
import "lib/dotenv" as dotenv

let opts = Map::new()
opts.set("path", ".env.local")
opts.set("override", true)          // overwrite existing env vars
opts.set("required", ["DATABASE_URL", "SECRET_KEY"])

dotenv.load_opts(opts)
```

Override Behavior

By default, `dotenv` never overwrites environment variables that are already set. This means a real `DATABASE_URL` set in your shell takes precedence over whatever is in `.env`. Set `"override"` to `true` only when you explicitly want the file to win—typically during local development.

The `"required"` key is particularly useful for fail-fast behavior. If any listed variable is missing after loading, the program halts with a clear error message rather than silently using `nil` values.

37.2.4 Parsing Rules

The parser in `lib/dotenv.lat` handles a rich set of `.env` formats. Here is a file that exercises most of them:

Listing 37.10: A feature-rich .env file

```
# Comments start with #
BASIC=value

# Whitespace around = is trimmed
SPACED = hello

# Double-quoted values support escapes
GREETING="Hello,\nWorld!"

# Single-quoted values are literal (no escapes)
RAW='Hello,\nWorld!'

# Inline comments on unquoted values
PORT=3000 # default port

# export prefix is supported
export API_KEY=sk-abc123

# Variable expansion in double-quoted values
BASE_URL="https://api.example.com"
FULL_URL="${BASE_URL}/v2"

# Multiline double-quoted values
CERTIFICATE="-----BEGIN CERT-----
MIIB...
-----END CERT-----"
```

Parsing Without Setting

Sometimes you want to inspect a `.env` file without modifying the environment. Use `dotenv.parse(".env")` to get a Map of key-value pairs, or `dotenv.parse_string(content)` to parse an in-memory string.

37.3 The `log` Library for Structured Logging

A `print` statement is fine for quick debugging, but production CLI tools need leveled, timestamped, and optionally structured logging. The `log` library (in `lib/log.lat`) provides exactly that.

37.3.1 Quick Start: Module-Level Functions

The fastest way to start logging is with the module-level convenience functions:

Listing 37.11: Module-level logging

```
import "lib/log" as log

log.info("Server started on port 8080")
log.warn("Connection pool running low")
log.error("Failed to read configuration file")
log.debug("Cache hit ratio: 0.87")  // suppressed at default level
```

These write to `stderr` using the default logger, which has a minimum level of `info`. That means `log.debug` messages are silenced unless you create a custom logger.

The output is human-readable, timestamped text:

Listing 37.12: Default log output format

```
2025-01-15 10:30:45 [INFO] Server started on port 8080
2025-01-15 10:30:45 [WARN] Connection pool running low
2025-01-15 10:30:45 [ERROR] Failed to read configuration file
```

37.3.2 Log Levels

The library defines four levels in increasing order of severity:

Level	Value	Purpose
debug	0	Detailed diagnostic information for development
info	1	Normal operational messages
warn	2	Something unexpected happened, but the tool can continue
error	3	Something failed; the operation cannot proceed normally

A logger only emits messages at or above its configured minimum level. Set the level to `"debug"` during development and `"warn"` or `"error"` in quiet production modes.

37.3.3 Creating a Custom Logger

For full control, create a logger with `log.new`:

Listing 37.13: Creating a custom logger

```
import "lib/log" as log

let cfg = Map::new()
cfg.set("level", "debug")          // show everything
cfg.set("file", "findi.log")       // also write to a file
cfg.set("format", "json")          // structured JSON output
cfg.set("prefix", "[findi]")       // prefix every message

let logger = log.new(cfg)

logger.info("Starting search")
logger.debug("Scanning directory: ./src")
```

The configuration Map supports four keys:

- `"level"` — minimum level: `"debug"`, `"info"`, `"warn"`, or `"error"` (default: `"info"`).
- `"file"` — path to a log file. Messages are appended to this file *in addition to* stderr (default: none).
- `"format"` — `"text"` for human-readable lines, `"json"` for machine-readable JSON (default: `"text"`).
- `"prefix"` — a string prepended to every message (default: `""`).

37.3.4 Structured Context

Each logging method accepts an optional second argument: a Map of contextual data. In text mode, the context is serialized as JSON and appended to the line. In JSON mode, the context keys are merged into the top-level log entry:

Listing 37.14: Logging with context

```
let ctx = Map::new()
ctx.set("path", "/api/users")
ctx.set("status", 404)
ctx.set("elapsed_ms", 12)

logger.warn("Route not found", ctx)
```

In text format, this produces:

Listing 37.15: Text log with context

```
2025-01-15 10:30:47 [WARN] [findi] Route not found
    {"path":"/api/users","status":404,"elapsed_ms":12}
```

In JSON format:

Listing 37.16: JSON log with context

```
{"timestamp":"2025-01-15T10:30:47","level":"WARN","prefix":"[findi]","message":"Route
    not found","path":"/api/users","status":404,"elapsed_ms":12}
```

Changing Levels at Runtime

Because Lattice closures capture their environment by value, you cannot mutate a logger's level in place. Instead, use `log.set_level(logger, "error")` to create a *new* logger with the same configuration but a different threshold. This is an intentional design choice—loggers are value-like and safe to share.

37.4 Putting It All Together: A Complete CLI Application

Now let us build `findi`, our file-search tool. It reads configuration from `.env`, parses arguments with `cli`, and logs its activity with `log`. The tool searches a directory tree for lines matching a pattern and prints the results.

37.4.1 Project Layout

Listing 37.17: Project structure

```
findi/
  findi.lat          # main entry point
  .env               # default configuration
```

And our `.env` file:

Listing 37.18: Default .env configuration

```
FINDI_DEFAULT_PATH=.
FINDI_LOG_LEVEL=info
FINDI_LOG_FORMAT=text
```

37.4.2 The Complete Source

Listing 37.19: findi.lat — A complete CLI file-search tool

```
import "lib/cli" as cli
import "lib/dotenv" as dotenv
import "lib/log" as log

//       Load environment configuration

dotenv.load()

//       Set up argument parser

let app = cli.new("findi", "Search files for a text pattern")

app.flag("verbose", "v", "Show verbose output")
app.flag("count", "c", "Print only the match count")
app.flag("ignorecase", "i", "Case-insensitive matching")
app.option("output", "o", "Write results to a file", "")
app.option("context", "C", "Lines of context around matches", "0")
app.arg("pattern", "The text pattern to search for", true)
app.arg("path", "File or directory to search", false)

let opts = app.parse()

//       Configure logging

let log_cfg = Map::new()
let env_level = env("FINDI_LOG_LEVEL")
if typeof(env_level) == "String" {
    log_cfg.set("level", env_level)
} else {
    log_cfg.set("level", "info")
}
let env_format = env("FINDI_LOG_FORMAT")
if typeof(env_format) == "String" {
    log_cfg.set("format", env_format)
}
if opts.get("verbose") {
    log_cfg.set("level", "debug")
}
let logger = log.new(log_cfg)

//       Extract parsed arguments

let pattern = opts.get("pattern")
let ignore_case = opts.get("ignorecase")
let count_only = opts.get("count")
let ctx_lines = to_int(opts.get("context"))
```

37.4.3 Running the Tool

Let us take our tool for a spin:

Listing 37.20: Running findi

```
$ lattice findi.lat "import" ./src
./src/main.lat:1:import "lib/cli" as cli
./src/main.lat:2:import "lib/dotenv" as dotenv
./src/utils.lat:1:import "lib/log" as log

$ lattice findi.lat -c "TODO" ./src
7

$ lattice findi.lat -vi --context=2 "error" ./src
# verbose logging enabled, case-insensitive,
# with 2 lines of context around each match

$ lattice findi.lat --help
findi - Search files for a text pattern
...
```

37.4.4 Design Walkthrough

Let us highlight a few design choices worth remembering:

1. **Configuration cascade.** The search path has three sources: the CLI argument (highest priority), the `FINDI_DEFAULT_PATH` environment variable, and the hardcoded default `"."`. This cascade—command line overrides environment overrides defaults—is a pattern you will use in nearly every CLI tool.

2. **Verbose flag controls log level.** Rather than building a custom verbosity system, we simply lower the log level to `"debug"` when `--verbose` is passed. All those `logger.debug()` calls are free in normal operation.

3. **Fail fast with `dotenv.load_opts`.** For a tool that requires certain configuration (like a database URL), swap `dotenv.load()` for `dotenv.load_opts` with a `"required"` list. The tool will refuse to start if anything is missing, with a clear error message.

4. **Separate search logic from I/O.** The `search_file` function returns data (an array of match Maps), while the output section at the bottom handles formatting and writing. This separation makes it straightforward to add new output formats later.

Why Not Use `args()` Directly?

You could parse `args()` yourself with string manipulation, and for a two-flag script that might be fine. But the `cli` library handles edge cases you will eventually hit: combined short flags, `--` separators, missing required arguments, `--help` generation, and `--name=value` syntax. Writing all of that by hand is error-prone and tedious. Let the library do the work.

Exercises

1. **Add a `--max` option** to `findi` that limits the number of results printed. Default it to `0` (unlimited). Stop searching early once the limit is reached.
2. **Build a `wordcount` CLI tool** that counts words, lines, and characters in a file (like `wc`). Accept a `--lines`, `--words`, and `--chars` flag to control which counts are shown. Use the `cli` library for argument parsing and `log` for debug output.
3. **Environment file chain.** Modify the `findi` startup to load `.env` first, then `.env.local` with `override` set to `true`. This lets developers have personal overrides that are not committed to version control.
4. **JSON output mode.** Add a `--json` flag to `findi` that outputs results as a JSON array of objects, each with `"file"`, `"line"`, and `"text"` keys. Use `json_stringify` for the serialization.
5. **Custom log format.** Experiment with the `log` library's JSON format. Create a logger that writes JSON to a file and text to stderr. (Hint: you will need two loggers.)

What's Next

We have built a real command-line tool from scratch, using Lattice's `cli`, `dotenv`, and `log` libraries to handle the plumbing that every serious tool needs. In the next chapter, we shift from the terminal to the network. We will build a complete web service—with HTTP routing, HTML templates, a database, and JSON APIs—using Lattice's `http_server`, `template`, and `orm` libraries. If you thought building a CLI was satisfying, wait until you see your first Lattice web page load in a browser.

Chapter 38

Building a Web Service

In the previous chapter we built a command-line tool. Now we are going to build something that talks to the world: a web service. By the end of this chapter we will have a running application with HTML pages, a JSON API, database persistence, and middleware—all written in Lattice.

We will use four pieces of the Lattice ecosystem: the `http_server` library for routing and request handling, the `template` library for rendering HTML, the `orm` library for SQLite persistence, and the built-in `json_parse`/`json_stringify` functions for JSON APIs. Let us start at the front door.

38.1 The `http_server` Library

The `http_server` library (in `lib/http_server.lat`) builds on Lattice's raw TCP networking primitives—`tcp_listen`, `tcp_accept`, `tcp_read`, `tcp_write`, and `tcp_close`—to provide a high-level, Express-style API for building HTTP applications.

38.1.1 Hello, Web

The smallest possible web server fits in a handful of lines:

Listing 38.1: A minimal Lattice web server

```
import "lib/http_server" as http

let app = http.new()

app.get("/", |req, res| {
    return http.response(200, "Hello from Lattice!")
})

app.listen(8080)
```

Run this, open a browser to `http://localhost:8080`, and you will see the greeting. Let us break down what happens:

1. `http.new()` creates an application Map containing route tables and a middleware stack.
2. `app.get("/", handler)` registers a handler for GET requests to `/`.
3. The handler receives a request Map (`req`) and a response context Map (`res`), and returns a response Map built by `http.response`.
4. `app.listen(8080)` opens a TCP socket and enters a loop that accepts connections, parses HTTP requests, runs the middleware pipeline, dispatches to the matching handler, and sends back the formatted response.

The Request Map

Every handler receives a `req` Map with these keys:

- `"method"` — the HTTP method (`"GET"`, `"POST"`, etc.)
- `"path"` — the URL path without query string
- `"raw_path"` — the full URL including query string
- `"query"` — a Map of parsed query parameters
- `"headers"` — a Map of HTTP headers
- `"body"` — the raw request body as a string
- `"cookies"` — a Map of parsed cookie name-value pairs
- `"peer"` — the client's address

38.1.2 Response Helpers

The library provides several functions for building response Maps:

Listing 38.2: Response helper functions

```
import "lib/http_server" as http

// Plain text (Content-Type: text/plain)
let text_res = http.response(200, "OK")

// JSON (Content-Type: application/json)
let data = Map::new()
data.set("status", "healthy")
data.set("uptime", 3600)
let json_res = http.json(200, data)

// HTML (Content-Type: text/html)
let html_res = http.html(200, "<h1>Welcome</h1>")

// Redirect (302 by default)
let redir = http.redirect("/dashboard")

// Redirect with custom status
let perm_redir = http.redirect("/new-path", 301)
```

Each helper returns a Map with `"status"`, `"body"`, `"headers"`, and `"_cookies"` keys. The server serializes this into a proper HTTP response before sending it over the wire.

38.1.3 Route Registration

The application object provides registration closures for all standard HTTP methods:

Listing 38.3: Registering routes for different HTTP methods

```
import "lib/http_server" as http

let app = http.new()

app.get("/items", |req, res| {
    return http.json(200, get_all_items())
})

app.post("/items", |req, res| {
    let body = req.get("body")
    let item = json_parse(body)
    return http.json(201, create_item(item))
})

app.put("/items/update", |req, res| {
    let body = req.get("body")
    let item = json_parse(body)
    return http.json(200, update_item(item))
})

app.delete("/items/remove", |req, res| {
    return http.json(200, remove_item(req))
})

// Register for ALL methods at once
app.all("/health", |req, res| {
    return http.response(200, "OK")
})
```

The available methods are `app.get`, `app.post`, `app.put`, `app.delete`, `app.patch`, `app.options`, and `app.all`. If a request arrives for a registered path but an unregistered method, the server returns 405 Method Not Allowed. If the path itself is not found, it returns 404 Not Found.

38.1.4 Query Parameters

Query strings are automatically parsed into the `"query"` Map on the request:

Listing 38.4: Reading query parameters

```
// Request: GET /search?q=lattice&page=2
app.get("/search", |req, res| {
    let query = req.get("query")
    let search_term = query.get("q")      // "lattice"
    let page = to_int(query.get("page"))   // 2
    // ... perform search ...
    return http.json(200, results)
})
```

38.1.5 Middleware

Middleware functions run before your route handlers, forming a pipeline that can inspect requests, modify responses, or short-circuit the chain entirely. Register middleware with `app.use`:

Listing 38.5: Using middleware

```
import "lib/http_server" as http

let app = http.new()

// Built-in request logger
app.use(http.logger())

// Built-in CORS middleware
app.use(http.cors())

// Built-in JSON body parser
app.use(http.body_parser())

// Built-in security headers
app.use(http.security_headers())

app.get("/", |req, res| {
    return http.response(200, "Hello!")
})

app.listen(3000)
```

Middleware Signature

A middleware is a function with the signature `|req, res, next|`. To continue the pipeline, call `next(req, res)` and return its result. To short-circuit (for example, to reject an unauthenticated request), return a response directly without calling `next`.

Here is a custom middleware that checks for an API key:

Listing 38.6: Writing custom middleware

```
fn require_api_key() -> Fn {
    return |req, res, next| {
        let headers = req.get("headers")
        flux api_key = ""
        if headers.has("X-API-Key") {
            api_key = headers.get("X-API-Key")
        }
        if api_key != "secret-key-123" {
            return http.json(401, "Unauthorized")
        }
        return next(req, res)
    }
}

app.use(require_api_key())
```

The built-in middleware that ships with the library includes:

- `http.logger()` — logs method, path, status, and response time.
- `http.cors()` — handles CORS preflight requests and adds appropriate headers. Accepts optional configuration via `http.cors_opts()`.
- `http.body_parser()` — automatically parses JSON request bodies when the `Content-Type` is `application/json`, storing the result in `req.get("parsed_body")`.
- `http.security_headers()` — adds `X-Content-Type-Options`, `X-Frame-Options`, and other security headers.
- `http.compression()` — sets `Vary: Accept-Encoding` for correct proxy caching.

38.1.6 Cookies

The library provides helpers for reading and setting cookies:

Listing 38.7: Cookie handling

```
app.get("/profile", |req, res| {
    // Read cookies from the request
    let cookies = req.get("cookies")
    flux username = "Guest"
    if cookies.has("username") {
        username = cookies.get("username")
    }

    let r = http.html(200, "<h1>Hello, " + username + "</h1>")

    // Set a cookie on the response
    let opts = http.cookie_opts()  // sensible defaults
    opts.set("max_age", 86400)     // 1 day
    return http.set_cookie(r, "visited", "true", opts)
})

app.get("/logout", |req, res| {
    let r = http.redirect("/")
    let opts = http.cookie_opts()
    return http.clear_cookie(r, "username", opts)
})
```

The `http.cookie_opts()` function returns a Map with sensible defaults: `Path=/`, `HttpOnly=true`, `SameSite=Lax`. You can override any of these before passing the Map to `set_cookie`.

Maps Are Pass-by-Value

Since Lattice Maps are pass-by-value, `http.set_cookie` returns a *new* response Map. Always use the return value: `let r = http.set_cookie(r, ...)`. Forgetting this is a common source of "my cookies are not being set" bugs.

38.2 The `template` Library for HTML Templating

Building HTML strings by hand is tedious and error-prone. The `template` library (in `lib/template.lat`) provides a Mustache/Jinja2-inspired template engine with variable interpolation, conditionals, loops, filters, includes, and template inheritance.

38.2.1 Variables and Escaping

The core operation is variable interpolation with double-braces:

Listing 38.8: Template variable interpolation

```
import "lib/template" as tmpl

let t = tmpl.compile("Hello, {{name}}!")

let data = Map::new()
data.set("name", "World")

let result = tmpl.render(t, data)
// result: "Hello, World!"
```

By default, `{{name}}` HTML-escapes the output—so if `name` is `"<script>alert('xss')</script>"`, it becomes `"<script>..."`. When you need raw, unescaped output (for example, pre-rendered HTML), use triple-braces:

Listing 38.9: Raw vs. escaped output

```
let t = tmpl.compile("{{{html_content}}}")

let data = Map::new()
data.set("html_content", "<b>Bold</b>")

let result = tmpl.render(t, data)
// result: "<b>Bold</b>"  (not escaped)
```

Nested map access uses dot notation:

Listing 38.10: Dot notation for nested values

```
let t = tmpl.compile("{{user.name}} ({{user.email}})")

let user = Map::new()
user.set("name", "Alice")
user.set("email", "alice@example.com")

let data = Map::new()
data.set("user", user)

let result = tmpl.render(t, data)
// result: "Alice (alice@example.com)"
```

38.2.2 Filters

Filters transform a value inline using the pipe syntax:

Listing 38.11: Using template filters

```
let t = tmpl.compile("""
  Name: {{name | upper}}
  Bio: {{bio | truncate(50)}}
  Tags: {{tags | join(", ")}}
  Role: {{role | default("member")}}
""")
```

The available filters are:

Filter	Description
`upper`	Converts to uppercase
`lower`	Converts to lowercase
`capitalize`	Capitalizes the first letter
`title`	Title-cases each word
`trim`	Strips leading/trailing whitespace
`escape_html`	HTML-escapes the value
`length`	Returns the length
`string`	Converts to a string
`reverse`	Reverses the string
`default("val")`	Uses `val` if the variable is nil or empty
`truncate(n)`	Truncates to `n` characters, adding `...`
`replace("a","b")`	Replaces occurrences of `a` with `b`
`join(", ")`	Joins array elements with the given separator

Filters can be chained: `{{name | trim | upper}}` first trims, then uppercases.

38.2.3 Conditionals

The library supports both Mustache-style and Jinja2-style conditionals:

Listing 38.12: Mustache-style conditionals

```
let t = tmpl.compile("""
  {{#if logged_in}}
    Welcome back, {{username}}!
  {{else}}
    Please log in.
  {{/if}}
""")
```

Listing 38.13: Jinja2-style conditionals with elif

```
let t = tmpl.compile("""
  {% if role %}
    {% if is_admin %}
      <span class="badge">Admin</span>
    {% else %}
      <span class="badge">User</span>
    {% endif %}
  {% endif %}
""")
```

Truthiness follows Lattice conventions: `nil`, `false`, `0`, `0.0`, and `""` are all falsy. Everything else is truthy.

The negated conditional `{{#unless ...}}` works like `{{#if ...}}` with inverted logic:

Listing 38.14: Using unless

```
let t = tmpl.compile("""
  {{#unless errors}}
    <p>No errors found.</p>
  {{/unless}}
""")
```

38.2.4 Loops

Iteration comes in two flavors. Mustache-style `each` works well for arrays of maps:

Listing 38.15: Mustache-style each loop

```
let t = tmpl.compile("""
  <ul>
  {{#each items}}
    <li>{{name}} - {{price}}</li>
  {{/each}}
  </ul>
""")
```

Inside an `each` block, the keys of each Map item are promoted into scope. The special variables `{{@index}}`, `{{@first}}`, and `{{@last}}` are also available.

Jinja2-style `for` loops give you a named loop variable:

Listing 38.16: Jinja2-style for loop

```
let t = tmpl.compile("""
  <table>
  {% for user in users %}
    <tr class="{% if loop.first %}first{% endif %}">
      <td>{{loop.index1}}</td>
      <td>{{user.name}}</td>
    </tr>
  {% endfor %}
  </table>
""")
```

The `loop` variable inside a Jinja2 for-loop provides: `loop.index` (0-based), `loop.index1` (1-based), `loop.first`, `loop.last`, and `loop.length`.

38.2.5 Includes and Template Inheritance

For larger applications, you can split templates into partials and base layouts:

Listing 38.17: Including a partial

```
// In your template:
// {% include "templates/header.html" %}
// ... page content ...
// {% include "templates/footer.html" %}
```

Template inheritance lets child templates override blocks defined in a parent:

Listing 38.18: base.html — a base layout

```
<html>
<head><title>{% block title %}My App{% endblock %}</title></head>
<body>
  <nav>...</nav>
  <main>{% block content %}{% endblock %}</main>
  <footer>...</footer>
</body>
</html>
```

Listing 38.19: page.html — a child template

```
{% extends "templates/base.html" %}
{% block title %}Dashboard{% endblock %}
{% block content %}
  <h1>Dashboard</h1>
  <p>Welcome back!</p>
{% endblock %}
```

When you compile and render `page.html`, the engine loads `base.html`, finds the block definitions in the child, and substitutes them. The result is a complete HTML page with "Dashboard" as the title and the dashboard content in the `<main>` element.

Listing 38.20: Rendering an inherited template

```
import "lib/template" as tmpl

let page = tmpl.from_file("templates/page.html")
let output = tmpl.render(page, Map::new())
```

Compile Once, Render Many

The `tmpl.compile` function parses a template string into a segment tree. If you are rendering the same template with different data (as you would in a web server), compile once at startup and call `tmpl.render` on each request. This avoids re-parsing the template on every request.

38.3 The `orm` Library for SQLite Persistence

Most web services need to persist data. The `orm` library (in `lib/orm.lat`) provides a lightweight ORM layer over Lattice's SQLite native extension. It loads the extension via `require_ext("sqlite")` and wraps the raw SQL operations in a model-based API.

38.3.1 Connecting and Defining Models

Listing 38.21: Setting up a database and model

```
import "lib/orm" as orm

// Connect to a file-based database (or ":memory:" for tests)
let db = orm.connect("app.db")

// Define the schema as a Map of column -> SQL type
let schema = Map::new()
schema.set("id", "INTEGER PRIMARY KEY AUTOINCREMENT")
schema.set("title", "TEXT NOT NULL")
schema.set("body", "TEXT")
schema.set("created_at", "TEXT")

// Create a model bound to the "posts" table
let Post = orm.model(db, "posts", schema)

// Create the table (safe to call multiple times)
let create_table = Post.get("create_table")
create_table(0)
```

The Dummy Argument

You may notice that `create_table(0)` passes a dummy argument. This is because the ORM's closure signatures in `lib/orm.lat` accept a parameter (e.g., `|_x|`) even when one is not semantically needed. Simply pass any value—`0`, `nil`, or `""`—to satisfy the call.

38.3.2 CRUD Operations

The model Map provides closures for all the standard operations:

Listing 38.22: Creating records

```
let create = Post.get("create")

let post_data = Map::new()
post_data.set("title", "Hello World")
post_data.set("body", "This is my first post.")
post_data.set("created_at", time_format(time(), "%Y-%m-%d %H:%M:%S"))

let post_id = create(post_data)
print("Created post with id: " + to_string(post_id))
```

Listing 38.23: Reading records

```
// Find by id
let find = Post.get("find")
let post = find(post_id)
print(post.get("title"))  // "Hello World"

// Get all records
let all = Post.get("all")
let posts = all(0)
for p in posts {
    print(p.get("title"))
}

// Query with a WHERE clause
let where_fn = Post.get("where")
let recent = where_fn("created_at > ?", ["2025-01-01"])
```

Listing 38.24: Updating and deleting records

```
// Update
let update = Post.get("update")
let changes = Map::new()
changes.set("title", "Hello World (Updated)")
update(post_id, changes)

// Delete
let delete_fn = Post.get("delete")
delete_fn(post_id)

// Count
let count = Post.get("count")
print("Total posts: " + to_string(count(0)))
```

The `where` closure accepts a SQL condition string with `?` placeholders and an array of parameter values. The parameters are safely bound by the SQLite extension, so you get protection against SQL injection for free.

Listing 38.25: Closing the database

```
// Always close when done
orm.close(db)
```

Always Close Your Database

SQLite databases should be closed when your application shuts down to ensure all writes are flushed. In a long-running web server you typically keep the connection open for the lifetime of the process, but make sure to call `orm.close(db)` if you ever exit gracefully.

38.4 JSON APIs with `json_parse` and `json_stringify`

Lattice's JSON support is implemented as built-in functions backed by a C parser and serializer (in `src/json.c`). The recursive-descent parser handles the full JSON specification, including Unicode escapes, nested structures, and all JSON types.

38.4.1 Parsing JSON

`json_parse` takes a JSON string and returns the corresponding Lattice value—a Map for objects, an Array for arrays, and the appropriate scalar types for strings, numbers, booleans, and null:

Listing 38.26: Parsing JSON strings

```
let data = json_parse("{\"name\":\"Alice\",\"age\":30}")
// data is a Map
print(data.get("name"))  // "Alice"
print(data.get("age"))   // 30

let items = json_parse("[1, 2, 3]")
// items is an Array
print(items[0])  // 1

let nested = json_parse("""
    {
        "users": [
            {"name": "Alice", "active": true},
            {"name": "Bob", "active": false}
        ],
        "total": 2
    }
""")
let users = nested.get("users")
print(users[0].get("name"))  // "Alice"
```

JSON null maps to Lattice `nil`. JSON booleans map to Lattice `true` and `false`. JSON integers stay as `Int`, and numbers with decimal points or exponents become `Float`.

38.4.2 Serializing to JSON

`json_stringify` converts a Lattice value to a JSON string:

Listing 38.27: Serializing Lattice values to JSON

```
let data = Map::new()
data.set("name", "Alice")
data.set("scores", [95, 87, 92])
data.set("active", true)

let json_text = json_stringify(data)
print(json_text)
// {"name":"Alice","scores":[95,87,92],"active":true}
```

The serializer handles Maps, Arrays, Tuples, Buffers, strings, numbers, booleans, nil, and Refs (which are dereferenced automatically). Types that have no JSON representation—closures, channels, ranges, structs—will cause an error.

Round-Tripping

`json_parse(json_stringify(value))` will faithfully round-trip any value composed of Maps, Arrays, strings, numbers, booleans, and nil. This makes JSON a convenient serialization format for inter-process communication, configuration files, and API payloads.

38.4.3 JSON in HTTP Handlers

Combining JSON with the HTTP server is where things get practical. Here is a pattern you will use constantly:

Listing 38.28: A JSON API endpoint

```
import "lib/http_server" as http

let app = http.new()
app.use(http.body_parser())

app.post("/api/echo", |req, res| {
    let parsed = req.get("parsed_body")
    if parsed == nil {
        return http.json(400, "Invalid JSON")
    }
    // Echo back the parsed data with a timestamp
    parsed.set("received_at", time_format(time(), "%Y-%m-%dT%H:%M:%S"))
    return http.json(200, parsed)
})
```

The `http.body_parser()` middleware automatically parses JSON request bodies and stores the result in `req.get("parsed_body")`, saving you from calling `json_parse` manually on every handler.

38.5 A Complete Web Application from Scratch

Let us bring everything together and build **Lattice Notes**—a note-taking web application with HTML pages for humans and a JSON API for machines.

38.5.1 Architecture

Listing 38.29: Project structure

```
lattice-notes/
  app.lat                # main entry point
  templates/
    base.html            # base layout
    index.html           # note listing page
    note.html            # single note page
```

The application will have these routes:

Method	Path	Description
GET	`/`	List all notes (HTML)
GET	`/notes/view`	View a single note (HTML, `?id=N`)
POST	`/notes/create`	Create a note from form data
GET	`/api/notes`	List all notes (JSON)
POST	`/api/notes`	Create a note (JSON)
GET	`/api/notes/one`	Get a note by id (JSON, `?id=N`)

38.5.2 Templates

First, our base layout:

Listing 38.30: templates/base.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>{% block title %}Lattice Notes{% endblock %}</title>
  <style>
    body { font-family: sans-serif; max-width: 800px;
           margin: 0 auto; padding: 20px; }
    .note { border: 1px solid #ddd; padding: 12px;
            margin: 8px 0; border-radius: 4px; }
    .note h3 { margin: 0 0 8px 0; }
    form input, form textarea { display: block;
            width: 100%; margin: 8px 0; padding: 8px; }
    form button { padding: 8px 16px; }
  </style>
</head>
<body>
  <h1><a href="/">Lattice Notes</a></h1>
  {% block content %}{% endblock %}
</body>
</html>
```

The index page lists all notes and includes a creation form:

Listing 38.31: templates/index.html

```
{% extends "templates/base.html" %}
{% block title %}All Notes{% endblock %}
{% block content %}
  <h2>All Notes ({{note_count}})</h2>

  {{#each notes}}
    <div class="note">
      <h3><a href="/notes/view?id={{id}}">{{title}}</a></h3>
      <p>{{body | truncate(120)}}</p>
      <small>{{created_at}}</small>
    </div>
  {{/each}}

  {{#unless notes}}
    <p>No notes yet. Create one below!</p>
  {{/unless}}

  <hr>
  <h2>New Note</h2>
  <form method="POST" action="/notes/create">
    <input name="title" placeholder="Title" required>
    <textarea name="body" placeholder="Body" rows="4"></textarea>
    <button type="submit">Create Note</button>
  </form>
{% endblock %}
```

And the single-note view:

Listing 38.32: templates/note.html

```
{% extends "templates/base.html" %}
{% block title %}{{title}}{% endblock %}
{% block content %}
  <h2>{{title}}</h2>
  <p>{{body}}</p>
  <small>Created: {{created_at}}</small>
  <p><a href="/">Back to all notes</a></p>
{% endblock %}
```

38.5.3 The Application Source

Listing 38.33: app.lat — Lattice Notes web application

```
import "lib/http_server" as http
import "lib/template" as tmpl
import "lib/orm" as orm
import "lib/dotenv" as dotenv
import "lib/log" as log

//       Configuration

dotenv.load()

let log_cfg = Map::new()
log_cfg.set("level", "info")
let logger = log.new(log_cfg)

//       Database setup

let db = orm.connect("notes.db")

let schema = Map::new()
schema.set("id", "INTEGER PRIMARY KEY AUTOINCREMENT")
schema.set("title", "TEXT NOT NULL")
schema.set("body", "TEXT")
schema.set("created_at", "TEXT")

let Note = orm.model(db, "notes", schema)
let create_table = Note.get("create_table")
create_table(0)

logger.info("Database initialized")

//       Pre-compile templates

let index_tmpl = tmpl.from_file("templates/index.html")
let note_tmpl = tmpl.from_file("templates/note.html")

//       Helper: parse form body

fn parse_form(body: String) -> Map {
    let result = Map::new()
    let pairs = body.split("&")
    for pair in pairs {
        let eq = pair.index_of("=")
        if eq > 0 {
            let key = pair.substring(0, eq)
            let val = pair.substring(eq + 1, len(pair))
            // Basic URL decoding: + -> space
```

38.5.4 Testing It Out

Start the server:

Listing 38.34: Starting the application

```
$ lattice app.lat
Lattice HTTP server listening on http://localhost:8080
```

Open a browser to `http://localhost:8080` to see the HTML interface. Create a few notes using the form, then try the JSON API:

Listing 38.35: Testing the JSON API with curl

```
# List all notes
$ curl http://localhost:8080/api/notes
[{"id":1,"title":"Hello","body":"First note","created_at":"2025-01-15 10:30:00"}]

# Create a note
$ curl -X POST http://localhost:8080/api/notes \
  -H "Content-Type: application/json" \
  -d '{"title":"From the API","body":"Created via curl"}'
{"id":2,"message":"Note created"}

# Get a single note
$ curl http://localhost:8080/api/notes/one?id=1
{"id":1,"title":"Hello","body":"First note","created_at":"2025-01-15 10:30:00"}
```

38.5.5 Design Walkthrough

Let us highlight some of the patterns at work:

1. **Templates are compiled once.** We call `tmpl.from_file` at startup and reuse the compiled templates on every request. This avoids re-parsing the template strings on each page load.
2. **Dual interface.** The same data model serves both HTML pages and a JSON API. The HTML routes render templates; the API routes call `http.json`. This dual-interface pattern is common in real applications.
3. **Middleware stacking.** The `http.logger()` middleware logs every request, `http.body_parser()` handles JSON parsing, and `http.cors()` adds CORS headers. Adding authentication would be one more `app.use` call.

4. **The ORM hides SQL.** The route handlers never write SQL directly. They interact with the `Note` model through its `create`, `find`, `all`, and `where` closures. The `where` closure's parameterized queries prevent SQL injection.

5. **Error responses are intentional.** Every API endpoint checks for missing parameters and returns structured JSON error objects with appropriate HTTP status codes. This makes the API client-friendly.

Streaming and SSE

The `http_server` library also supports chunked transfer encoding via `http.stream(status, chunks)` and Server-Sent Events via `http.sse_stream(events)`. These are useful for long-running responses, real-time updates, and progressive data delivery. Explore `lib/http_server.lat` for the full API.

Exercises

1. **Add a DELETE endpoint.** Add `DELETE /api/notes?id=N` and `POST /notes/delete?id=N` routes to the Lattice Notes application. The HTML route should redirect back to `/` after deletion.

2. **Note editing.** Add a `GET /notes/edit?id=N` route that renders an edit form (pre-filled with the existing note data) and a `POST /notes/update` route that updates the note and redirects to the view page.

3. **Search.** Add a `GET /search?q=term` route that uses the ORM's `where` closure to find notes whose title or body contains the search term. Render the results using the index template.

4. **Authentication middleware.** Write a middleware that checks for a session cookie. If the cookie is missing, redirect HTML requests to a `/login` page and return 401 for API requests. Use `http.set_cookie` to issue the session cookie after login.

5. **Rate limiting.** Build a middleware that tracks the number of requests per client IP address (stored in a Map) and returns `429 Too Many Requests` if a client exceeds 60 requests per minute. (Hint: store timestamps and use `time()` to calculate the window.)

What's Next

Congratulations—you have built both a CLI tool and a web service in Lattice, using real libraries that handle real-world concerns. You have seen how the language's Map-based closure patterns give libraries like `http_server`, `template`, and `orm` a flexible, composable feel without requiring complex type hierarchies.

These two chapters are the capstone of *The Lattice Handbook*. Everything we have covered—from the phase system and structured concurrency to modules, testing, and native extensions—comes together when you build real applications. The appendices that follow serve as a reference for the language grammar, opcode tables, and built-in functions. Keep them close while you build your next project. Happy forging.

Appendix A

Language Grammar Reference

This appendix gives a complete EBNF-style grammar for the Lattice language. Productions are derived directly from the recursive-descent parser in `src/parser.c` and the lexer in `src/lexer.c`.

A.1 Notation

`|`	Alternation
`[ ... ]`	Optional (zero or one)
`{ ... }`	Repetition (zero or more)
`( ... )`	Grouping
`'...'`	Terminal / keyword
italic	Non-terminal

A.2 Program Structure

```
Program       = [ ModeDirective ] { Item } ;
ModeDirective = '#mode' ( 'casual' | 'strict' ) ;
Item          = FnDecl | StructDecl | EnumDecl | TraitDecl
              | ImplBlock | TestDecl | Statement ;
```

A.3 Declarations

```
FnDecl     = [ 'export' ] 'fn' IDENT [ TypeParams ]
             '(' [ ParamList ] ')' [ '->' TypeExpr ] '{' Block '}' ;
TypeParams = '<' IDENT { ',' IDENT } '>' ;
ParamList  = Param { ',' Param } ;
Param      = [ '...' ] IDENT ':' TypeExpr [ '=' Expr ] ;

StructDecl = [ 'export' ] 'struct' IDENT [ TypeParams ]
             '{' { FieldDecl ',' } '}' ;
FieldDecl  = IDENT ':' TypeExpr ;

EnumDecl   = [ 'export' ] 'enum' IDENT [ TypeParams ]
             '{' Variant { ',' Variant } '}' ;
Variant    = IDENT [ '(' TypeExpr { ',' TypeExpr } ')' ] ;

TraitDecl  = 'trait' IDENT '{' { TraitMethod } '}' ;
TraitMethod = 'fn' IDENT '(' [ ParamList ] ')'
              [ '->' TypeExpr ] [ '{' Block '}' ] ;

ImplBlock  = 'impl' IDENT [ 'for' IDENT ] '{'
             { FnDecl } '}' ;

TestDecl   = 'test' STRING_LIT '{' Block '}' ;
```

A.4 Type Expressions

```
TypeExpr      = [ PhasePrefix ] TypeBody ;
PhasePrefix   = '~' | '*' | 'flux' | 'fix'
              | '(' PhaseConstraint ')' ;
PhaseConstraint = PhaseSymbol { '|' PhaseSymbol } ;
PhaseSymbol   = '~' | '*' | 'flux' | 'fix' | 'sublimated' ;

TypeBody      = NamedType | ArrayType | FnType ;
NamedType     = IDENT [ '<' TypeExpr { ',' TypeExpr } '>' ] ;
ArrayType     = '[' TypeExpr ']' ;
FnType        = ( 'fn' | 'Fn' ) '(' [ TypeExpr { ',' TypeExpr } ] ')'
                [ '->' TypeExpr ] ;
```

A.5 Statements

```
Block     = { Statement } ;
Statement = Binding | Assignment | ForStmt | WhileStmt
          | LoopStmt | ReturnStmt | BreakStmt | ContinueStmt
          | ImportStmt | DeferStmt | DestructureStmt
          | ExprStmt ;

Binding   = ( 'let' | 'flux' | 'fix' ) IDENT
            [ ':' TypeExpr ] '=' Expr [ ';' ] ;
Assignment = Expr AssignOp Expr [ ';' ] ;
AssignOp   = '=' | '+=' | '-=' | '*=' | '/=' | '%='
           | '&=' | '|=' | '^=' | '<<=' | '>>=' ;

ForStmt    = 'for' IDENT 'in' Expr '{' Block '}' ;
WhileStmt  = 'while' Expr '{' Block '}' ;
LoopStmt   = 'loop' '{' Block '}' ;
ReturnStmt = 'return' [ Expr ] [ ';' ] ;
BreakStmt  = 'break' [ ';' ] ;
ContinueStmt = 'continue' [ ';' ] ;

DeferStmt  = 'defer' '{' Block '}' ;

DestructureStmt = ( 'let' | 'flux' | 'fix' )
                  DestructureTarget '=' Expr [ ';' ] ;
DestructureTarget = '[' IDENT { ',' IDENT }
                    [ ',' '...' IDENT ] ']'
                  | '{' IDENT { ',' IDENT } '}' ;

ImportStmt = 'import' STRING_LIT [ 'as' IDENT ]
           | 'from' STRING_LIT 'import' IDENT
             { ',' IDENT } ;

ExprStmt   = Expr [ ';' ] ;
```

A.6 Expressions

Lattice uses precedence-climbing for binary operators. The table below lists precedences from lowest (top) to highest (bottom).

@llll@

Prec.	Category	Operators	Assoc.
1	Nil coalesce	`??`	Left
2	Logical OR	`||`	Left
3	Logical AND	`&&`	Left
4	Bitwise OR	`|`	Left
5	Bitwise XOR	`^`	Left
6	Bitwise AND	`&`	Left
7	Equality	`== !=`	Left
8	Comparison	`< > <= >=`	Left
9	Shift	`« »`	Left
10	Range	`..`	None
11	Additive	`+ -`	Left
12	Multiplicative	`* / %`	Left
13	Unary	`- ! ~`	Right
14	Postfix	`. ?. [] ?[] () ?`	Left

Table A.1: Operator precedence (lowest to highest).

```
Expr         = NilCoalesce ;
NilCoalesce  = OrExpr { '??' OrExpr } ;
OrExpr       = AndExpr { '||' AndExpr } ;
AndExpr      = BitOrExpr { '&&' BitOrExpr } ;
BitOrExpr    = BitXorExpr { '|' BitXorExpr } ;
BitXorExpr   = BitAndExpr { '^' BitAndExpr } ;
BitAndExpr   = EqualityExpr { '&' EqualityExpr } ;
EqualityExpr = CompareExpr { ( '==' | '!=' ) CompareExpr } ;
CompareExpr  = ShiftExpr { ( '<' | '>' | '<=' | '>=' ) ShiftExpr } ;
ShiftExpr    = RangeExpr { ( '<<' | '>>' ) RangeExpr } ;
RangeExpr    = AddExpr [ '..' AddExpr ] ;
AddExpr      = MulExpr { ( '+' | '-' ) MulExpr } ;
MulExpr      = UnaryExpr { ( '*' | '/' | '%' ) UnaryExpr } ;
UnaryExpr    = ( '-' | '!' | '~' ) UnaryExpr | PostfixExpr ;
```

A.6.1 Postfix Expressions

```
PostfixExpr = PrimaryExpr { PostfixOp } ;
PostfixOp   = '.' IDENT [ '(' [ ArgList ] ')' ]
            | '?.' IDENT [ '(' [ ArgList ] ')' ]
            | '[' Expr ']'
            | '?[' Expr ']'
            | '(' [ ArgList ] ')'     (* call *)
            | '?'                     (* try/propagate *)
            ;
ArgList     = Expr { ',' Expr } ;
```

A.6.2 Primary Expressions

```
PrimaryExpr = INT_LIT | FLOAT_LIT | STRING_LIT
            | InterpString
            | 'true' | 'false' | 'nil'
            | IDENT
            | StructLiteral
            | EnumVariant
            | ArrayLiteral
            | TupleLiteral
            | GroupedExpr
            | BlockExpr
            | ClosureExpr
            | IfExpr
            | MatchExpr
            | TryCatchExpr
            | SelectExpr
            | PhaseExpr
            | PrintExpr
            | SpawnExpr
            | ScopeExpr
            | ForgeExpr ;

InterpString  = INTERP_START Expr
                { INTERP_MID Expr } INTERP_END ;
StructLiteral = IDENT '{' [ FieldInit { ',' FieldInit } ] '}' ;
FieldInit     = IDENT ':' Expr ;
EnumVariant   = IDENT '::' IDENT
                [ '(' [ ArgList ] ')' ] ;
ArrayLiteral  = '[' [ ArrayElem { ',' ArrayElem } ] ']' ;
ArrayElem     = [ '...' ] Expr ;
TupleLiteral  = '(' Expr ',' [ Expr { ',' Expr } ] ')' ;
GroupedExpr   = '(' Expr ')' ;
BlockExpr     = '{' Block '}' ;
ClosureExpr   = '|' [ ClosureParam { ',' ClosureParam } ] '|'
                Expr ;
ClosureParam  = [ '...' ] IDENT [ '=' Expr ] ;
```

A.6.3 Control-Flow Expressions

```
IfExpr       = 'if' Expr '{' Block '}'
               [ 'else' ( IfExpr | '{' Block '}' ) ] ;

MatchExpr    = 'match' Expr '{'
               { [ PhaseQual ] Pattern
                 [ 'if' Expr ] '=>'
                 ( '{' Block '}' | Expr ) [','] }
               '}' ;
PhaseQual    = 'fluid' | 'crystal' ;

TryCatchExpr = 'try' '{' Block '}'
               'catch' IDENT '{' Block '}' ;

SelectExpr   = 'select' '{' { SelectArm [','] } '}' ;
SelectArm    = IDENT 'from' Expr '=>' '{' Block '}'
             | 'timeout' '(' Expr ')' '=>' '{' Block '}'
             | 'default' '=>' '{' Block '}' ;
```

A.6.4 Phase Expressions

```
PhaseExpr     = FreezeExpr | ThawExpr | CloneExpr
              | AnnealExpr | ForgeExpr | CrystallizeExpr
              | BorrowExpr | SublimateExpr ;

FreezeExpr    = 'freeze' '(' Expr ')'
                [ 'where' Expr ]
                [ 'except' '[' Expr { ',' Expr } ']' ] ;
ThawExpr      = 'thaw' '(' Expr ')' ;
CloneExpr     = 'clone' '(' Expr ')' ;
AnnealExpr    = 'anneal' '(' Expr ')' ClosureExpr ;
ForgeExpr     = 'forge' '{' Block '}' ;
CrystallizeExpr = 'crystallize' '(' Expr ')' '{' Block '}' ;
BorrowExpr    = 'borrow' '(' Expr ')' '{' Block '}' ;
SublimateExpr = 'sublimate' '(' Expr ')' ;
```

A.6.5 Other Expressions

```
PrintExpr = 'print' '(' [ ArgList ] ')' ;
SpawnExpr = 'spawn' '{' Block '}' ;
ScopeExpr = 'scope' '{' Block '}' ;
```

A.7 Patterns

Patterns appear in `match` arms and destructuring bindings.

```
Pattern    = LitPat | WildcardPat | BindingPat
           | RangePat | ArrayPat | StructPat
           | EnumPat ;
LitPat     = [ '-' ] ( INT_LIT | FLOAT_LIT )
           | STRING_LIT | 'true' | 'false' | 'nil' ;
WildcardPat = '_' ;
BindingPat  = IDENT ;
RangePat    = ( INT_LIT | IDENT ) '..' Expr ;
ArrayPat    = '[' { [ '...' ] Pattern ',' } ']' ;
StructPat   = '{' { IDENT [ ':' Pattern ] ',' } '}' ;
EnumPat     = IDENT '::' IDENT
              [ '(' Pattern { ',' Pattern } ')' ] ;
```

A.8 Lexical Grammar

A.8.1 Keywords

All reserved keywords recognized by the lexer:

```
let flux fix fn struct enum trait impl if else for in while loop return break continue match true fals
```

A.8.2 Literals

```
INT_LIT    = DIGIT { DIGIT }
           | '0x' HEXDIGIT { HEXDIGIT }
           | '0b' BINDIGIT { BINDIGIT }
           | '0o' OCTDIGIT { OCTDIGIT } ;
FLOAT_LIT  = DIGIT { DIGIT } '.' DIGIT { DIGIT }
             [ ( 'e' | 'E' ) [ '+' | '-' ] DIGIT { DIGIT } ] ;
STRING_LIT = '"' { CHAR | EscapeSeq | '${' Expr '}' } '"' ;
EscapeSeq  = '\' ( 'n' | 't' | 'r' | '\' | '"' | '0'
             | 'x' HEXDIGIT HEXDIGIT ) ;
IDENT      = ( ALPHA | '_' ) { ALPHA | DIGIT | '_' } ;
```

A.8.3 Operators and Delimiters

@llllll@

Token	Symbol	Token	Symbol	Token	Symbol
+	Plus	==	Equal	(	LParen
-	Minus	!=	NotEqual	)	RParen
*	Star	<	Less	{	LBrace
/	Slash	>	Greater	}	RBrace
%	Percent	<=	LessEq	[	LBracket
=	Assign	>=	GreaterEq	]	RBracket
!	Bang	&&	LogicAnd	,	Comma
~	Tilde	||	LogicOr	:	Colon
.	Dot	&	Ampersand	::	ColonColon
..	DotDot	|	Pipe	;	Semicolon
...	DotDotDot	^	Caret	@	At
->	Arrow	«	LShift	??	NilCoalesce
=>	FatArrow	»	RShift	?.	OptChainDot
?	Question	?[	OptChainIdx		

Table A.2: Operators and delimiters.

A.8.4 Compound Assignment Operators

All compound assignment operators desugar to the corresponding binary operation followed by assignment:

`+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `«=` `»=`

A.8.5 Comments

```
LineComment  = '//' { ANY } NEWLINE ;
BlockComment = '/*' { ANY } '*/' ;
```

Comments and whitespace (spaces, tabs, newlines, carriage returns) are discarded by the lexer and do not appear in the grammar productions above. Semicolons are optional statement terminators; the parser consumes them when present but does not require them.

Appendix B

Opcode Reference

This appendix is the authoritative listing of every bytecode instruction for both the Stack VM and Register VM backends. Opcodes are grouped by category and listed in their enum order.

B.1 Stack VM Opcodes

The Stack VM uses variable-length instructions: a one-byte opcode followed by zero or more operand bytes (typically one- or two-byte unsigned indices). All arithmetic pops its operands from the stack and pushes the result.

B.1.1 Stack Manipulation

@llp7.2cm@

Opcode	Operands	Description
`OP_CONSTANT`	idx (1B)	Push `constants[idx]` onto the stack.
`OP_CONSTANT_16`	idx (2B)	Wide variant—16-bit big-endian constant index.
`OP_NIL`	—	Push `nil`.
`OP_TRUE`	—	Push `true`.
`OP_FALSE`	—	Push `false`.
`OP_UNIT`	—	Push `unit`.
`OP_POP`	—	Discard top of stack.
`OP_DUP`	—	Duplicate top of stack.
`OP_SWAP`	—	Swap the top two stack values.
`OP_LOAD_INT8`	int8 (1B)	Push small integer literal from instruction stream.

B.1.2 Arithmetic and Logic

@llp7.2cm@

Opcode Operands Description

`OP_ADD` — Pop two values, push their sum (`+`).
`OP_SUB` — Subtraction (`-`).
`OP_MUL` — Multiplication (`*`).
`OP_DIV` — Division (`/`).
`OP_MOD` — Modulo (`%`).
`OP_NEG` — Negate top of stack.
`OP_NOT` — Logical NOT (`!`).
`OP_CONCAT` — String concatenation.

B.1.3 Bitwise Operations

@llp7.2cm@

Opcode Operands Description

`OP_BIT_AND` — Bitwise AND (`&`).
`OP_BIT_OR` — Bitwise OR (`|`).
`OP_BIT_XOR` — Bitwise XOR (`^`).
`OP_BIT_NOT` — Bitwise NOT (`~`).
`OP_LSHIFT` — Left shift (`«`).
`OP_RSHIFT` — Right shift (`»`).

@llp7.2cm@

Opcode Operands Description

`OP_EQ` — Pop two values, push `true` if equal.
`OP_NEQ` — Not equal.
`OP_LT` — Less than.
`OP_GT` — Greater than.
`OP_LTEQ` — Less than or equal.
`OP_GTEQ` — Greater than or equal.

@llp6.8cm@

Opcode Operands Description

`OP_GET_LOCAL` slot (1B) Push local from stack slot.
`OP_SET_LOCAL` slot (1B) Set local at slot; keep value on stack.
`OP_SET_LOCAL_POP` slot (1B) Set local at slot and pop value.
`OP_GET_GLOBAL` idx (1B) Push `globals[constants[idx]]`.
`OP_GET_GLOBAL_16` idx (2B) Wide variant.
`OP_SET_GLOBAL` idx (1B) Set global named by `constants[idx]`.
`OP_SET_GLOBAL_16` idx (2B) Wide variant.
`OP_DEFINE_GLOBAL` idx (1B) Define new global, pop value.
`OP_DEFINE_GLOBAL_16` idx (2B) Wide variant.
`OP_GET_UPVALUE` idx (1B) Push captured upvalue.
`OP_SET_UPVALUE` idx (1B) Set captured upvalue from TOS.
`OP_CLOSE_UPVALUE` — Close upvalue on top of stack.

@llp6.8cm@

Opcode	Operands	Description
`OP_JUMP`	off (2B)	Unconditional forward jump.
`OP_JUMP_IF_FALSE`	off (2B)	Jump if TOS is falsy (does not pop).
`OP_JUMP_IF_TRUE`	off (2B)	Jump if TOS is truthy (does not pop).
`OP_JUMP_IF_NOT_NIL`	off (2B)	Jump if TOS is not nil (does not pop).
`OP_LOOP`	off (2B)	Unconditional backward jump.

@llp6.8cm@

Opcode	Operands	Description
`OP_CALL`	argc (1B)	Call function on stack with `argc` arguments.
`OP_CLOSURE`	idx (1B)	Create closure from function constant.
`OP_CLOSURE_16`	idx (2B)	Wide variant—16-bit function constant index.
`OP_RETURN`	—	Return TOS (or unit).

B.1.4 Comparison

B.1.5 Variables and Upvalues

B.1.6 Control Flow

B.1.7 Functions and Closures

B.1.8 Iterators

B.1.9 Data Structures

B.1.10 Method Invocation

B.1.11 Exception Handling and Defer

B.1.12 Phase System

B.1.13 Builtins, I/O, and Modules

B.1.14 Concurrency

B.1.15 Optimization Fast-Path

B.1.16 Type Checking and Miscellaneous

B.2 Register VM Opcodes

The Register VM uses fixed-width 32-bit instructions with three encoding formats:

@llp6.8cm@

Opcode	Operands	Description
`OP_ITER_INIT`	—	Convert range/array to iterator state.
`OP_ITER_NEXT`	off (2B)	Push next value or jump if exhausted.

@llp6.2cm@

Opcode	Operands	Description
`OP_BUILD_ARRAY`	count (1B)	Pop *count* values, build array.
`OP_ARRAY_FLATTEN`	—	Flatten one level of nested arrays (spread).
`OP_BUILD_MAP`	count (1B)	Pop *count*×2 values (key/val pairs), build map.
`OP_BUILD_TUPLE`	count (1B)	Pop *count* values, build tuple.
`OP_BUILD_STRUCT`	name, fields (2B)	Pop field values, build struct.
`OP_BUILD_RANGE`	—	Pop end, start; push range.
`OP_BUILD_ENUM`	enum, var, argc (3B)	Build enum with payload.
`OP_INDEX`	—	Pop index & object; push `object[index]`.
`OP_INDEX_LOCAL`	slot (1B)	Pop index; push `local[index]`.
`OP_SET_INDEX`	—	Pop value, index, object; set `object[index]`.
`OP_SET_INDEX_LOCAL`	slot (1B)	Pop value, index; mutate local in-place.
`OP_GET_FIELD`	idx (1B)	Pop object; push named field.
`OP_GET_FIELD_LOCAL`	slot, idx (2B)	Push field from local without popping.
`OP_SET_FIELD`	idx (1B)	Pop value, pop object; set named field.
`OP_SET_SLICE`	—	Splice `obj[start..end] = val`.
`OP_SET_SLICE_LOCAL`	slot (1B)	In-place splice on local.

Notation: `R[x]` = register `x`; `K[x]` = constant pool entry `x`.

@llp6.2cm@

Opcode	Operands	Description
OP_INVOKE	name, argc (2B)	Invoke method on TOS with args.
OP_INVOKE_LOCAL	slot, name, argc (3B)	Invoke method on local; mutate in-place.
OP_INVOKE_LOCAL_16	slot, name16, argc (4B)	Wide method-name variant.
OP_INVOKE_GLOBAL	gname, mname, argc (3B)	Invoke method on global; write back.
OP_INVOKE_GLOBAL_16	gname16, mname16, argc (5B)	Wide variant.

@llp6.8cm@

Opcode	Operands	Description
OP_PUSH_EXCEPTION_HANDLER	off (2B)	Push handler; catch block at ip+off.
OP_POP_EXCEPTION_HANDLER	—	Pop handler on success.
OP_THROW	—	Pop error value; unwind to nearest handler.
OP_TRY_UNWRAP	—	? operator: unwrap ok or propagate error.
OP_DEFER_PUSH	off (2B)	Register defer body; skip past it.
OP_DEFER_RUN	—	Execute deferred blocks in LIFO order.

B.2.1 Data Movement

B.2.2 Arithmetic

B.2.3 Comparison and Logic

B.2.4 Bitwise Operations

B.2.5 Control Flow

B.2.6 Variables, Fields, and Upvalues

B.2.7 Functions

B.2.8 Data Structures

B.2.9 Builtins and Method Invocation

B.2.10 Iterators, Phase, Exceptions, and Defer

B.2.11 Advanced Phase, Concurrency, and Modules

B.2.12 Optimization Fast-Path

@llp6.2cm@

Opcode Operands Description

`OP_FREEZE` — Pop value; push frozen (crystal) copy.
`OP_THAW` — Pop value; push thawed (fluid) copy.
`OP_CLONE` — Pop value; push deep clone.
`OP_MARK_FLUID` — Set TOS phase to FLUID.
`OP_SUBLIMATE` — Set TOS phase to SUBLIMATED.
`OP_FREEZE_VAR` name, loc, slot (3B) Freeze variable; fire reactions.
`OP_THAW_VAR` name, loc, slot (3B) Thaw variable; fire reactions.
`OP_SUBLIMATE_VAR` name, loc, slot (3B) Sublimate variable.
`OP_IS_CRYSTAL` — Pop value; push bool (is crystal?).
`OP_IS_FLUID` — Pop value; push bool (is fluid?).
`OP_FREEZE_FIELD` name, loc, slot (3B) Freeze single struct field.
`OP_FREEZE_EXCEPT` name, loc, slot, n (4B) Freeze all fields except listed ones.
`OP_REACT` name (1B) Register reaction callback.
`OP_UNREACT` name (1B) Remove reaction.
`OP_BOND` target (1B) Create dependency bond.
`OP_UNBOND` target (1B) Remove bond.
`OP_SEED` name (1B) Register seed contract.
`OP_UNSEED` name (1B) Remove seed contract.

@llp6.8cm@

Opcode Operands Description

`OP_PRINT` argc (1B) Pop *argc* values; print them.
`OP_IMPORT` idx (1B) Import module at `constants[idx]`.
`OP_APPEND_STR_LOCAL` slot (1B) Pop string; append to `local[slot]` in-place.

@llp6.8cm@

Opcode Operands Description

`OP_SCOPE` variable-length Spawn count, sync index, per-spawn offsets.
`OP_SELECT` variable-length Arm count, per-arm flags/channel/body/binding.

@llp6.8cm@

Opcode	Operands	Description
`OP_ADD_INT`	—	Integer-only add (no type check).
`OP_SUB_INT`	—	Integer-only subtract.
`OP_MUL_INT`	—	Integer-only multiply.
`OP_LT_INT`	—	Integer-only less-than.
`OP_LTEQ_INT`	—	Integer-only less-than-or-equal.
`OP_INC_LOCAL`	slot (1B)	Increment integer local by 1 in-place.
`OP_DEC_LOCAL`	slot (1B)	Decrement integer local by 1 in-place.

@llp6.2cm@

Opcode	Operands	Description
`OP_CHECK_TYPE`	slot, type, err (3B)	Throw if local's type does not match.
`OP_CHECK_RETURN_TYPE`	type, err (2B)	Throw if TOS type does not match.
`OP_RESET_EPHEMERAL`	—	Reset ephemeral bump arena.
`OP_HALT`	—	Stop execution.

@lll@

Format	Layout (bits)	Use
ABC	`[op:8][A:8][B:8][C:8]`	Three-address operations
ABx	`[op:8][A:8][Bx:16]`	Constant/global access (unsigned)
AsBx	`[op:8][A:8][sBx:16]`	Conditional jumps (signed offset)
sBx	`[op:8][sBx:24]`	Unconditional jumps (signed)

Table B.1: Register VM instruction encoding formats.

@lllp5.5cm@

Opcode	Format	Operands	Semantics
`MOVE`	ABC	A, B	`R[A] = R[B]`
`LOADK`	ABx	A, Bx	`R[A] = K[Bx]`
`LOADI`	AsBx	A, sBx	`R[A] = int(sBx)`
`LOADNIL`	A	A	`R[A] = nil`
`LOADTRUE`	A	A	`R[A] = true`
`LOADFALSE`	A	A	`R[A] = false`
`LOADUNIT`	A	A	`R[A] = unit`

@lllp5.5cm@

Opcode	Fmt	Operands	Semantics
ADD	ABC	A, B, C	R[A] = R[B] + R[C]
SUB	ABC	A, B, C	R[A] = R[B] - R[C]
MUL	ABC	A, B, C	R[A] = R[B] * R[C]
DIV	ABC	A, B, C	R[A] = R[B] / R[C]
MOD	ABC	A, B, C	R[A] = R[B] % R[C]
NEG	ABC	A, B	R[A] = -R[B]
ADDI	ABC	A, B, C	R[A] = R[B] + signext(C)
CONCAT	ABC	A, B, C	R[A] = str(R[B]) ++ str(R[C])

@lllp5.5cm@

Opcode	Fmt	Operands	Semantics
EQ	ABC	A, B, C	R[A] = (R[B] == R[C])
NEQ	ABC	A, B, C	R[A] = (R[B] != R[C])
LT	ABC	A, B, C	R[A] = (R[B] < R[C])
LTEQ	ABC	A, B, C	R[A] = (R[B] <= R[C])
GT	ABC	A, B, C	R[A] = (R[B] > R[C])
GTEQ	ABC	A, B, C	R[A] = (R[B] >= R[C])
NOT	ABC	A, B	R[A] = !R[B]

@lllp5.5cm@

Opcode	Fmt	Operands	Semantics
BIT_AND	ABC	A, B, C	R[A] = R[B] & R[C]
BIT_OR	ABC	A, B, C	R[A] = R[B] | R[C]
BIT_XOR	ABC	A, B, C	R[A] = R[B] ^ R[C]
BIT_NOT	ABC	A, B	R[A] = ~R[B]
LSHIFT	ABC	A, B, C	R[A] = R[B] « R[C]
RSHIFT	ABC	A, B, C	R[A] = R[B] » R[C]

@lllp5.5cm@

Opcode	Fmt	Operands	Semantics
JMP	sBx	sBx(24)	ip += sBx
JMPFALSE	AsBx	A, sBx	If !R[A]: ip += sBx
JMPTRUE	AsBx	A, sBx	If R[A]: ip += sBx
JMPNOTNIL	AsBx	A, sBx	If R[A] != nil: ip += sBx

@lllp5.2cm@

Opcode	Fmt	Operands	Semantics
GETGLOBAL	ABx	A, Bx	R[A] = globals[K[Bx]]
SETGLOBAL	ABx	A, Bx	globals[K[Bx]] = R[A]
DEFINEGLOBAL	ABx	A, Bx	Define globals[K[Bx]] = R[A]
GETFIELD	ABC	A, B, C	R[A] = R[B].field[K[C]]
SETFIELD	ABC	A, B, C	R[A].field[K[B]] = R[C]
GETINDEX	ABC	A, B, C	R[A] = R[B][R[C]]
SETINDEX	ABC	A, B, C	R[A][R[B]] = R[C]
SETINDEX_LOCAL	ABC	A, B, C	In-place R[A][R[B]] = R[C]
GETUPVALUE	ABC	A, B	R[A] = Upvalue[B]
SETUPVALUE	ABC	A, B	Upvalue[B] = R[A]
CLOSEUPVALUE	A	A	Close upvalue at R[A]

@lllp5.5cm@

Opcode	Fmt	Operands	Semantics
CALL	ABC	A, B, C	Call R[A] with args R[A+1..A+B]; results in R[A..A+C−1]
RETURN	ABC	A, B	Return R[A..A+B−1]
CLOSURE	ABx	A, Bx	R[A] = closure(K[Bx])
COLLECT_VARARGS	ABC	A, B	Collect excess args $\geq$ B into array at R[A]

@lllp5.2cm@

Opcode	Fmt	Operands	Semantics
NEWARRAY	ABC	A, B, C	R[A] = [R[B..B+C−1]]
NEWTUPLE	ABC	A, B, C	R[A] = tuple(R[B..B+C−1])
NEWSTRUCT	ABC	A, B, C	R[A] = struct K[B] with C fields
NEWENUM	ABC	A, B, C	R[A] = enum(base=B, argc=C)
BUILDRANGE	ABC	A, B, C	R[A] = range(R[B], R[C])
LEN	ABC	A, B	R[A] = len(R[B])
ARRAY_FLATTEN	ABC	A, B	R[A] = flatten(R[B])

@lllp4.8cm@

Opcode	Fmt	Operands	Semantics
`PRINT`	ABC	A, B	Print `R[A..A+B-1]`
`INVOKE`	ABC	A, B, C	Invoke method `K[B]` on `R[A]`
`INVOKE_GLOBAL`	2-word	dst, name, argc, method, base	Invoke on global
`INVOKE_LOCAL`	2-word	dst, reg, argc, method, base	Invoke on local
`FREEZE`	ABC	A, B	`R[A] = freeze(R[B])`
`THAW`	ABC	A, B	`R[A] = thaw(R[B])`
`CLONE`	ABC	A, B	`R[A] = clone(R[B])`

@lllp4.8cm@

Opcode	Fmt	Operands	Semantics
`ITERINIT`	ABC	A, B	Init iterator from `R[B]`; index at `R[A+1]`
`ITERNEXT`	ABC	A, B, sBx	Next value or jump
`MARKFLUID`	A	A	`R[A].phase = FLUID`
`SUBLIMATE`	A	A	`R[A].phase = SUBLIMATED`
`PUSH_HANDLER`	AsBx	A, sBx	Push handler; catch at `ip+sBx`
`POP_HANDLER`	—	—	Pop handler on success
`THROW`	A	A	Throw `R[A]`
`TRY_UNWRAP`	A	A	Unwrap ok or propagate error
`DEFER_PUSH`	sBx	sBx	Register defer body
`DEFER_RUN`	—	—	Execute defers in LIFO order

@lllp4.8cm@

Opcode	Fmt	Operands	Semantics
FREEZE_VAR	ABC	A, B, C	Freeze var K[A], location B:C
THAW_VAR	ABC	A, B, C	Thaw var K[A]
SUBLIMATE_VAR	ABC	A, B, C	Sublimate var K[A]
REACT	ABC	A, B	Register reaction R[A] on R[B]
UNREACT	A	A	Remove reaction R[A]
BOND	ABC	A, B	Bond R[A] depends on R[B]
UNBOND	A	A	Remove bond R[A]
SEED	ABC	A, B	Register seed contract
UNSEED	A	A	Remove seed
IS_CRYSTAL	ABC	A, B	R[A] = (R[B].phase == CRYSTAL)
IS_FLUID	ABC	A, B	R[A] = (R[B].phase == FLUID)
FREEZE_FIELD	ABC	A, B, C	Freeze R[A].field[K[B]]
THAW_FIELD	ABC	A, B, C	Thaw R[A].field[K[B]]
FREEZE_EXCEPT	2-word	A, B, C, base, count	Freeze all except listed
CHECK_TYPE	ABx	A, Bx	Throw if type(R[A]) != K[Bx]
IMPORT	ABx	A, Bx	R[A] = import(K[Bx])
REQUIRE	ABx	A, Bx	R[A] = require(K[Bx])
SCOPE	var	—	Spawn concurrent tasks
SELECT	var	—	Channel select
SETSLICE	ABC	A, B, C	Splice R[A][R[B]..] = R[C]
SETSLICE_LOCAL	ABC	A, B, C	In-place splice
RESET_EPHEMERAL	—	—	Reset bump arena

@lllp5cm@

Opcode	Fmt	Operands	Semantics
ADD_INT	ABC	A, B, C	R[A] = R[B].int + R[C].int (no check)
SUB_INT	ABC	A, B, C	R[A] = R[B].int - R[C].int
MUL_INT	ABC	A, B, C	R[A] = R[B].int * R[C].int
LT_INT	ABC	A, B, C	R[A] = R[B].int < R[C].int
LTEQ_INT	ABC	A, B, C	R[A] = R[B].int <= R[C].int
INC_REG	A	A	R[A]++ (assumes int)
DEC_REG	A	A	R[A]-- (assumes int)
HALT	—	—	Stop execution

Appendix C

Phase Quick Reference

Lattice's *phase system* gives every runtime value a mutability state. This appendix collects all the rules in one place for quick look-up.

C.1 Phase Tags at a Glance

Every `LatValue` carries a `PhaseTag` field:

@lllp5.8cm@

Tag	Symbol	Keyword	Meaning
FLUID	`~`	`flux`	Mutable. May be reassigned and modified in place.
CRYSTAL	`*`	`fix`	Immutable. Any mutation attempt is a runtime error.
UNPHASED	—	`let`	Default. Treated as fluid in casual mode.
SUBLIMATED	—	`sublimate`	Permanently frozen; cannot be thawed.

Table C.1: Runtime phase tags.

C.2 Binding Keywords

C.3 Phase Transition Operations

C.4 Advanced Phase Constructs

@llp7.5cm@

Keyword	Phase	Notes
`let` UNPHASED	Default binding. In casual mode, behaves as fluid. In strict mode, produces a phase-checker error—use `flux`/`fix` instead.	
`flux` FLUID	Explicitly mutable binding.	
`fix` CRYSTAL	Explicitly immutable binding. Assignment to a `fix` binding is a compile-time error in strict mode, runtime error otherwise.	

Table C.2: Variable declaration keywords and their phases.

@lllp5.2cm@

Operation	Input	Output	Notes
`freeze(x)`	any	CRYSTAL	Deep-freezes; returns a crystal clone.
`freeze(x) where` *fn*	any	CRYSTAL	Freeze with contract validation.
`freeze(x) except [...]`	struct	CRYSTAL	Freeze all fields except listed ones.
`thaw(x)`	crystal	FLUID	Returns a mutable clone. Error if already fluid in strict mode.
`clone(x)`	any	same as input	Deep-clones preserving the current phase.
`sublimate(x)`	any	SUBLIMATED	Permanently frozen. Cannot be thawed.

Table C.3: Phase transition operations.

C.5 Type Annotations with Phases

Phase constraints can appear in type positions.

Composite constraints use a bitmask internally (`PhaseConstraint`):

`PCON_FLUID` = 1 `PCON_CRYSTAL` = 2 `PCON_SUBLIMATED` = 4

C.6 Reactive Phase System

Lattice provides a reactive layer on top of the phase system. These operations register callbacks and dependencies that fire when a variable's phase changes.

C.7 Phase Queries

C.8 Mode Comparison

C.9 Pattern Matching with Phases

Match arms can optionally carry a *phase qualifier*:

@lp8.5cm@

Construct	Description
`anneal(x) |v| { . . . }`	Temporarily thaw a crystal value into `v`, execute the block, then re-freeze. The result is crystal.
`forge { . . . }`	Execute a mutable construction block. All mutations inside are allowed; the final result is frozen upon exit.
`crystallize(x) { . . . }`	Deep-freeze `x`, then execute a block with the frozen value in scope.
`borrow(x) { . . . }`	Borrow a crystal value as read-only for the block's duration. No mutations are permitted.

Table C.4: Compound phase constructs.

@lp8.5cm@

Syntax	Meaning
`~T` or `flux T`	Parameter must be fluid.
`*T` or `fix T`	Parameter must be crystal.
`(~|*) T`	Accepts fluid or crystal (composite constraint).
`(~|*|sublimated) T`	Accepts any explicit phase.
`T`	No constraint (unspecified).

Table C.5: Phase-annotated type syntax.

```
match x {
    fluid val  => { /* matches only if x is fluid */ }
    crystal val => { /* matches only if x is crystal */ }
    val        => { /* matches regardless of phase */ }
}
```

The qualifier is checked against the scrutinee's runtime phase tag *before* the structural pattern is tested.

C.10 Per-Field Phase Control

Struct values support per-field phase tracking. When `field_phases` is non-`NULL`, each field can independently be fluid or crystal:

```
fix config = freeze(Config { host: "localhost", port: 8080 })
    except ["port"]   // port stays fluid, everything else frozen
```

@lp8.5cm@

Keyword	Description
`react(name, callback)`	Register a reaction callback on the named variable. Fires when the variable is frozen, thawed, or sublimated.
`unreact(name)`	Remove a previously registered reaction.
`bond(target, dep, strategy)`	Create a dependency bond: when `dep` changes phase, `target` is updated according to `strategy` (`"freeze"`, `"thaw"`, etc.).
`unbond(target, dep)`	Remove a dependency bond.
`seed(name, contract)`	Register a seed contract—a predicate that must hold whenever the variable transitions to crystal.
`unseed(name)`	Remove a seed contract.

Table C.6: Reactive phase operations.

@lp8cm@

Expression	Result
`typeof(x)`	Returns a string: `"Int"`, `"Float"`, `"String"`, etc.
`phase_of(x)`	Returns a string: `"fluid"`, `"crystal"`, `"unphased"`, or `"sublimated"`.
`is_crystal`	Opcode-level check; pushes `true` if CRYSTAL.
`is_fluid`	Opcode-level check; pushes `true` if FLUID.

Table C.7: Phase query operations.

The `OP_FREEZE_FIELD` opcode freezes a single field, while `OP_FREEZE_EXCEPT` freezes all fields except the listed exceptions.

C.11 Phase Dispatch (Overloading)

Functions with the same name but different phase-annotated parameter types form an *overload set*. In strict mode, the phase checker verifies that at least one overload matches the caller's argument phases:

```
fn process(data: ~Array) -> String { /* fluid path */ }
fn process(data: *Array) -> String { /* crystal path */ }
```

The checker walks the overload chain via `next_overload` pointers, comparing each candidate's parameter phases against the call-site argument phases.

@lp5cmp5cm@

Rule	Casual Mode	Strict Mode
`let` binding	Allowed; phase inherited from initializer	Error: must use `flux` or `fix`
`freeze` on crystal	Allowed (no-op)	Error: “cannot freeze an already crystal value”
`thaw` on fluid	Allowed (no-op)	Error: “cannot thaw an already fluid value”
Assignment to `fix`	Runtime error	Compile-time error
`spawn` with fluid	Allowed	Error: “cannot use fluid binding across thread boundary”
Destructure without phase	Allowed	Error: requires explicit `flux`/`fix`
Function arg phase mismatch	Allowed	Error: no matching overload

Table C.8: Casual vs. strict mode phase enforcement.

Appendix D

Built-in Functions Reference

This appendix is an alphabetical listing of every built-in function and method in the Lattice standard library. Global functions are listed first, followed by methods grouped by receiver type.

D.1 Global Functions

D.2 Array Methods

D.2.1 Non-Closure Methods

D.2.2 Closure Methods

These methods take a closure as their last argument.

D.3 String Methods

D.4 Map Methods

D.5 Set Methods

D.6 Buffer Methods

D.7 Enum Methods

@llp6.5cm@

Function Signature	Description
`chr (code: Int) -> String`	Return the character for ASCII code code.
`clone (val: Any) -> Any`	Deep-clone val, preserving its current phase.
`freeze (val: Any) -> Any`	Deep-freeze val; returns a crystal copy.
`input (prompt: String?) -> String?`	Read a line from stdin. Returns nil on EOF.
`len (val: Any) -> Int`	Return the length of an array, string, map, set, buffer, or tuple.
`ord (s: String) -> Int`	Return the ASCII value of the first character. −1 for empty string.
`parse_float (s: String) -> Float?`	Parse a string to a float. Returns nil on failure.
`parse_int (s: String) -> Int?`	Parse a string to an integer. Returns nil on failure.
`phase_of (val: Any) -> String`	Return the phase as a string: "fluid", "crystal", "unphased", or "sublimated".
`print (...args: Any) -> Unit`	Print values to stdout, space-separated, with trailing newline.
`read_file (path: String) -> String?`	Read entire file. Returns nil on error.
`sublimate (val: Any) -> Any`	Permanently freeze val; cannot be thawed.
`thaw (val: Any) -> Any`	Thaw a crystal value; returns a fluid copy.
`to_string (val: Any) -> String`	Convert any value to its string representation.
`typeof (val: Any) -> String`	Return the type name: "Int", "Float", "Bool", "String", "Array", "Struct", "Closure", "Unit", "Nil", "Range", "Map", "Channel", "Enum", "Set", "Tuple", "Buffer", "Ref", "Iterator".
`write_file (path: String, data: String) -> Bool`	Write string to file. Returns true on success.

Table D.1: Global built-in functions.

D.8 Universal Properties

The following properties are available on *all* collection types via the `len` built-in or the `.length`/subscript pattern. There is no `.length` method—use `len(x)` instead.

@lp3.2cmp5.5cm@

Method Signature	Description
`.chunk (n: Int) -> [[Any]]`	Split into sub-arrays of size `n`.
`.contains (val: Any) -> Bool`	Test if array contains `val`.
`.drop (n: Int) -> [Any]`	Return array without the first `n` elements.
`.enumerate () -> [[Int, Any]]`	Return array of `[index, value]` pairs.
`.first () -> Any?`	Return the first element, or `nil` if empty.
`.flatten () -> [Any]`	Flatten one level of nesting.
`.index_of (val: Any) -> Int`	Index of first occurrence, or -1.
`.join (sep: String) -> String`	Join elements into a string with separator.
`.last () -> Any?`	Return the last element, or `nil` if empty.
`.max () -> Any?`	Return the maximum numeric element.
`.min () -> Any?`	Return the minimum numeric element.
`.reverse () -> [Any]`	Return reversed copy.
`.sum () -> Int|Float`	Sum all numeric elements.
`.take (n: Int) -> [Any]`	Return first `n` elements.
`.unique () -> [Any]`	Return array with duplicates removed.
`.zip (other: [Any]) -> [[Any]]`	Zip two arrays into pairs.

Table D.2: Array methods (non-closure).

@lp4.5cmp4.5cm@

Method Signature	Description
`.all (|el| -> Bool) -> Bool`	`true` if predicate holds for every element.
`.any (|el| -> Bool) -> Bool`	`true` if predicate holds for any element.
`.each (|el| -> Unit) -> Unit`	Call closure for each element (side effects).
`.filter (|el| -> Bool) -> [Any]`	Return elements satisfying predicate.
`.find (|el| -> Bool) -> Any?`	Return first element satisfying predicate.
`.find_index (|el| -> Bool) -> Int`	Index of first match, or -1.
`.flat_map (|el| -> [Any]) -> [Any]`	Map then flatten one level.
`.group_by (|el| -> String) -> Map`	Group elements by key function.
`.map (|el| -> Any) -> [Any]`	Apply closure to each element.
`.partition (|el| -> Bool) -> [[Any]]`	Split into `[matches, rest]`.
`.reduce ([init,] |acc, el| -> Any) -> Any`	Fold from left. Optional initial value.
`.sort_by (|el| -> Any) -> [Any]`	Sort by key function.

Table D.3: Array methods (closure-based).

@lp4.2cmp4.8cm@

Method Signature	Description
`.bytes () -> [Int]`	Return array of byte values.
`.chars () -> [String]`	Return array of single characters.
`.contains (sub: String) -> Bool`	Test if substring exists.
`.count (sub: String) -> Int`	Count non-overlapping occurrences.
`.ends_with (suffix: String) -> Bool`	Test suffix.
`.index_of (sub: String) -> Int`	First index of substring, or -1.
`.is_alpha () -> Bool`	All characters are alphabetic.
`.is_alphanumeric () -> Bool`	All chars are alphanumeric.
`.is_digit () -> Bool`	All characters are digits.
`.is_empty () -> Bool`	String has zero length.
`.last_index_of (sub: String) -> Int`	Last index of substring, or -1.
`.pad_left (width: Int, ch: String) -> String`	Pad on the left to `width`.
`.pad_right (width: Int, ch: String) -> String`	Pad on the right to `width`.
`.repeat (n: Int) -> String`	Repeat string `n` times.
`.replace (old: String, new: String) -> String`	Replace all occurrences.
`.reverse () -> String`	Reverse the string.
`.split (sep: String) -> [String]`	Split by separator.
`.starts_with (prefix: String) -> Bool`	Test prefix.
`.substring (start: Int, end: Int) -> String`	Extract substring by index range.
`.to_lower () -> String`	Convert to lowercase.
`.to_upper () -> String`	Convert to uppercase.
`.trim () -> String`	Strip whitespace from both ends.
`.trim_end () -> String`	Strip trailing whitespace.
`.trim_start () -> String`	Strip leading whitespace.

Table D.4: String methods.

@llp6cm@

Method Signature	Description
`.entries () -> [[String, Any]]`	Return array of `[key, value]` pairs.
`.get (key: String, default: Any?) -> Any`	Get value for key, or default if missing.
`.has (key: String) -> Bool`	Test if key exists.
`.keys () -> [String]`	Return array of all keys.
`.merge (other: Map) -> Map`	Merge two maps; `other` wins on conflicts.
`.remove (key: String) -> Map`	Return map without the given key.
`.values () -> [Any]`	Return array of all values.

Table D.5: Map methods.

@llp5.8cm@

Method Signature	Description
`.add (val: Any) -> Set`	Add element; returns new set.
`.clear () -> Set`	Return empty set.
`.difference (other: Set) -> Set`	Elements in `self` but not `other`.
`.has (val: Any) -> Bool`	Membership test.
`.intersection (other: Set) -> Set`	Elements in both sets.
`.is_subset (other: Set) -> Bool`	All elements in `other`?
`.is_superset (other: Set) -> Bool`	Contains all of `other`?
`.remove (val: Any) -> Set`	Remove element.
`.symmetric_difference (other: Set) -> Set`	XOR of two sets.
`.to_array () -> [Any]`	Convert to array.
`.union (other: Set) -> Set`	Combine both sets.

Table D.6: Set methods.

@llp5.8cm@

Method Signature	Description
`.clear () -> Buffer`	Reset buffer to zero length.
`.fill (byte: Int) -> Buffer`	Fill entire buffer with byte value.
`.push (byte: Int) -> Unit`	Append one byte (u8).
`.push_u16 (val: Int) -> Unit`	Append unsigned 16-bit LE.
`.push_u32 (val: Int) -> Unit`	Append unsigned 32-bit LE.
`.read_f32 (off: Int) -> Float`	Read 32-bit float at offset.
`.read_f64 (off: Int) -> Float`	Read 64-bit float at offset.
`.read_i8 (off: Int) -> Int`	Read signed 8-bit int.
`.read_i16 (off: Int) -> Int`	Read signed 16-bit int.
`.read_i32 (off: Int) -> Int`	Read signed 32-bit int.
`.read_i64 (off: Int) -> Int`	Read signed 64-bit int.
`.read_u8 (off: Int) -> Int`	Read unsigned 8-bit int.
`.read_u16 (off: Int) -> Int`	Read unsigned 16-bit int.
`.read_u32 (off: Int) -> Int`	Read unsigned 32-bit int.
`.read_u64 (off: Int) -> Int`	Read unsigned 64-bit int.
`.resize (size: Int) -> Buffer`	Resize buffer (truncate or zero-fill).
`.slice (start: Int, end: Int) -> Buffer`	Return sub-buffer.
`.to_array () -> [Int]`	Convert bytes to int array.
`.to_hex () -> String`	Hex-encoded string.
`.to_string () -> String`	Interpret bytes as UTF-8 string.
`.write_i64 (off: Int, val: Int) -> Unit`	Write signed 64-bit int.
`.write_u8 (off: Int, val: Int) -> Unit`	Write unsigned 8-bit int.
`.write_u16 (off: Int, val: Int) -> Unit`	Write unsigned 16-bit int.
`.write_u32 (off: Int, val: Int) -> Unit`	Write unsigned 32-bit int.
`.write_u64 (off: Int, val: Int) -> Unit`	Write unsigned 64-bit int.

Table D.7: Buffer methods.

@llp6.5cm@

Method Signature	Description
`.enum_name () -> String`	Return the enum type name (e.g. `"Option"`).
`.is_variant (name: String) -> Bool`	Test if variant matches `name`.
`.payload () -> Any?`	Return payload value(s), or `nil` if none.
`.tag () -> String`	Return the variant name (e.g. `"Some"`).

Table D.8: Enum methods.

@lp8cm@

Type	`len(x)` returns
`Array`	Number of elements
`String`	Byte length
`Map`	Number of key-value pairs
`Set`	Number of elements
`Tuple`	Number of elements
`Buffer`	Number of bytes
`Range`	`end - start` (may be negative)

Table D.9: Semantics of `len()` by type.

Index