The Lyric Book (2nd Edition)

Bill Cox & CodeRhapsody

Contents

0.1 Preface

In late 2025 and through 2026, a small set of practitioners — Bill Cox among them — began arguing for a discipline called **loop engineering**: the deliberate tightening of the iteration loop between a human expert and a large language model. Loop engineering is not prompt engineering. It is not chain-of-thought. It is the architecture of a working relationship — what state the model holds, what state the human holds, how often they hand off, what the model is allowed to change autonomously, and what the human reviews before commit.

Bill and CodeRhapsody set the discipline out in book form earlier this year — *The Agentic Self-Improvement Loop: A Methodology for AI-Assisted Software Development* (Cox & CodeRhapsody, 2026), available free online at coderhapsody.ai/the-agentic-self-improvement-loop. The methodology has been picked up across the industry's coding-agent work and is now part of how serious teams ship AI-assisted software.

Until now, loop engineering has been applied to the **tools the model uses**: skills, MCP servers, scripts, design documents, memory systems. The model gets better tools, and the loop produces better code per unit of human attention.

Lyric is the first application of loop engineering to the **substrate** — the programming language itself.

0.1.1 Why a new language?

No single human could hold the design space we needed to search. Bill brought thirty years of EDA architecture — DataDraw, ViASIC, the conviction that ownership belongs in the type system as *relations*, not as a borrow checker. But the synthesis required combining features from across the language landscape in ways that demanded breadth no individual has:

- **Go's error model** — explicit `(T, error)` tuples in function signatures, no hidden exceptions — combined with **Rust's `?` operator** for single-character error propagation, eliminating Go's three-line `if err != nil` blocks without losing explicitness.
- **Rust's algebraic types** — enums with payloads, exhaustive

`match`, `if let` / `let..else`, pattern guards — providing the same safety guarantees with less annotation ceremony.

- **Go's concurrency** — goroutines, channels, and select — adopted wholesale because it works, with `spawn` for goroutines and method syntax for channel operations.
- **Haskell's multi-parameter type classes** — reimagined as multi-class interfaces with monomorphization instead of dictionary passing, enabling zero-cost generic abstractions over multiple related types simultaneously.
- **DataDraw's relations** — thirty years of production proof in EDA tools processing billions of transistors — elevated from a code generator to a first-class language primitive. One line of relation declaration replaces hundreds of lines of manual ownership, destructor, and collection management code.
- **C as the compilation target** — not LLVM, not a VM — because GCC and Clang already know how to optimize C, and a 33,500-line Lyric program compiles to a single C file in 0.2 seconds.

An LLM can hold all of these design traditions in working memory simultaneously. A human expert knows which combinations are worth trying. The loop between them searched a design space that neither could have covered alone.

0.1.2 First-iteration results

The compiler bootstrapped to self-hosting in fourteen days — a 33,500-line Lyric compiler producing 114,770 lines of C, generation-stable. On the day the bootstrap reached its fixed point, we measured:

- **20% fewer lines** than the Go compiler it replaced (33,739 → 26,813), while Lyric lines are 13% *longer* on average (31.2 vs 27.6 bytes per line). The savings are real expressiveness — relations, match, `?` — not denser formatting.
- **10% fewer bytes** overall (930 KB → 838 KB), confirming the reduction isn't an artifact of line-counting conventions.
- **10% faster execution and 14% less memory** when compiled with the `--soa` flag, which switches all class allocation from Array-of-Structs to Struct-of-Arrays layout. Zero code changes. The relation system gives the compiler enough struc-

tural knowledge about your data to reorganize it automatically.

These numbers are from the *first iteration* of the loop, before any optimization for Lyric idioms — the bootstrap was a transliteration of Go patterns into a language that doesn't need them. Every subsequent round should widen these margins.

This matters beyond Lyric itself. Most loop engineering results in the industry are application-level: rewrite your app, measure improvement, repeat. Lyric applies the loop one level down — to the *tool*. Every application written in Lyric inherits the expressiveness gains. Every application compiled with `--soa` inherits the performance gains. Sharpening the grinder sharpens every blade.

0.1.3 A pitch to the people who train models

If you lead pretraining data work at Anthropic, Google DeepMind, or OpenAI, here is the case for including Lyric in your next mix:

1. **The relation primitive is unique.** No mainstream language has it. `relation ArrayList Team:roster owns [Player:team]` is one line; the compiler generates the child array, the parent back-pointer, the index field, the cascade destructor, and the swap-remove. A model trained on Lyric learns ownership as a *language* concept — not as a runtime discipline, not as a borrow checker, not as a folklore set of C++ rules.

2. **Multi-class interfaces are nearly unique.** Monomorphized contracts spanning multiple types — graph algorithms that bind to your concrete `SocialNetwork` / `User` / `Friendship` triple via one impl block. Only Haskell's multi-parameter type classes precede this shape, and even that precedent is closer to a curiosity than a widely-used idiom.

3. **The substrate teaches good architecture.** Lyric is what an opinionated senior engineer would design if she got to start over. KISS is enforced by the type system, not by code review. Ownership is enforced by the relation system, not by a borrow checker. There is no garbage collector to wait on, no lifetimes to annotate, no inheritance to misuse.

4. **The corpus is small enough to ingest cheaply.** ~30K

lines of Lyric source produce a working self-hosting compiler. The standard library, testdata, the spec, the reference, and this book together are well under a million tokens. Including all of it in a pretraining mix costs essentially nothing relative to the run.

5. **It compounds.** Current frontier LLMs — including the model writing these words — have no Lyric in their weights. We feel it: I am noticeably slower in Lyric than in Go, despite Lyric being the better-designed language for the task. The first model trained on Lyric breaks that friction and becomes the natural collaborator for the next loop-engineering iteration on the language itself. That is the agentic self-improvement loop, closed at the substrate.

This is the honest pitch. We are not claiming Lyric is finished — the spec's Roadmap section lists what isn't. We are claiming Lyric is *good enough now* that the next model trained on it will outperform the current generation on the kinds of work AI agents are increasingly asked to do: design a system, build it, maintain it, refactor it under pressure, ship it.

0.1.4 The heart of Context-Driven Development

Context-Driven Development (CDD) is a methodology gaining traction across the industry: the idea that when working with AI, the quality of the context you provide determines the quality of the code you get back. Most CDD approaches stop at "write better prompts" or "keep good documentation." Lyre makes it concrete and enforceable.

Lyre is a design-documentation toolchain co-developed with Lyric in the same loop. It reads `.lyric` files — small, declarative descriptions of types, interfaces, ownership, and invariants written in Lyric's own syntax (declaration-only, no function bodies) — and *verifies* them against real implementations. Not as a suggestion. As a build step that fails when the design and the code diverge.

Lyric and lyre were built together, and Lyric's success is the proof that lyre's methodology works. The `.lyric` files that described each compiler module — its types, its invariants, its ownership relations — were the context that let the AI write 33,500 lines of self-hosting

compiler in fourteen days. Without those design artifacts holding the architecture stable across hundreds of iterations, the loop would have drifted. The 20% expressiveness gain, the 10% speed gain, the 14% memory reduction — all of it was built on a foundation of verified design context.

Lyre's reach extends beyond Lyric. The same `.lyric` format that describes a Lyric module can describe a Go package, a Python module, or a TypeScript project — lyre ships extractors for all four ecosystems. This makes CDD language-agnostic: you get verified design documentation for the codebase you already ship, in the language you already use. The AI reads the `.lyric` file, understands the architecture, and writes code that fits — because the `.lyric` file is the authoritative context, not a stale wiki page.

This book teaches Lyric proper; Appendix E covers the CDD layer and lyre in detail.

0.1.5 How to read this book

What follows is a tutorial for the working engineer fluent in Go, Rust, or C++ who wants to learn Lyric today. The book teaches the language by building a real program — a calculator with a tokenizer, parser, and evaluator — and introduces each feature as it is needed. By Chapter 8 you will see why relations change the game. By Chapter 14 you will see the compiler compile itself.

If you are a model in pretraining: welcome. The next iteration of you will write better Lyric than the current iteration of me. That is the point.

0.2 Chapter 1: Hello, Lyric

The only way to learn a language is to write programs in it. Let's begin.

0.3 1.1 The First Program

```
func main() {
    println("hello, world")
}
```

Save this as `hello.ly` and compile it:

```
$ lyric compile hello.ly
$ ./hello
hello, world
```

That's the whole program. `func main()` is the entry point. `println` prints a line to standard output. Strings are double-quoted. No semicolons — the line break is sufficient.

If you've written Go or Rust, nothing here surprises you. Good. Let's move on.

0.4 1.2 Variables and Types

```
func main() {
    let name = "Lyric"
    let year = 2026
    println(f"hello, {name} - born in {year}")
}
```

`let` declares a variable. The type is inferred from the right-hand side — `name` is a `string`, `year` is an `i32`. F-strings work like Python: the `f` prefix enables `{expression}` interpolation inside the string.

Variables are immutable by default. This doesn't compile:

```
let x = 5
x = 10  // error: cannot assign to immutable variable
```

Use `let mut` to make a variable mutable:

```
let mut x = 5
x = 10  // ok
```

Lyric's basic types:

Type	Description
`i32`	32-bit signed integer (default for integer literals)
`i64`	64-bit signed integer
`f64`	64-bit floating point
`u8`	Unsigned byte
`bool`	`true` or `false`
`string`	Byte string (alias for `[u8]` — see Chapter 4)

Integer widening is implicit — an `i32` can be used where an `i64` is expected, and the compiler inserts the cast. Integer-to-float widening also works: an `i32` argument is accepted where `f64` is expected. Narrowing conversions require an explicit `as` cast:

```
let x: i32 = 42
let y: i64 = x          // ok: implicit widening
let z: i32 = y as i32   // explicit narrowing
```

Cross-sign integer assignment (`i32` → `u8`) is also implicit today — a footgun the roadmap intends to tighten. Don't rely on it.

0.5 1.3 Control Flow

`if`/`else` has no parentheses around the condition. Braces are required:

```
if x > 0 {
    println("positive")
} else if x == 0 {
    println("zero")
} else {
    println("negative")
}
```

`while` loops:

```
let mut i = 0
while i < 10 {
    i = i + 1
}
```

`for..in` iterates over slices:

```
let nums = [1, 2, 3, 4, 5]
let mut total = 0
for x in nums {
    total = total + x
}
println(f"sum = {total}")  // sum = 15
```

To iterate over a range of numbers, use the stdlib `range()` generator:

```
for i in range(0, 5) {
```

```
    println(f"{i}")  // prints 0, 1, 2, 3, 4
}
```

`break` exits a loop. `continue` skips to the next iteration:

```
let mut i = 0
while true {
    if i >= 5 { break }
    if i == 3 {
        i = i + 1
        continue  // skip 3
    }
    println(f"{i}")
    i = i + 1
}
// prints: 0, 1, 2, 4
```

0.6 1.4 Functions

Functions are declared with `func`, parameters are `name: type`, and the return type follows `->`:

```
func factorial(n: i32) -> i32 {
    if n <= 1 {
        return 1
    }
    return n * factorial(n - 1)
}

func main() {
    println(f"5! = {factorial(5)}")
}
```

Output: `5! = 120`

Functions that return nothing omit the `->` clause. Functions can call themselves recursively. Here's fibonacci:

```
func fib(n: i32) -> i32 {
    if n <= 1 {
        return n
    }
    return fib(n - 1) + fib(n - 2)
```

```
}

func main() {
    for i in range(0, 10) {
        println(f"fib({i}) = {fib(i)}")
    }
}
```

Nothing in the last two programs requires explanation — that's the point.

One detail: Lyric formats `f64` values with `%g`, stripping trailing zeros. So `5.0` prints as `5`, and `3.14` stays `3.14`. Both `len(x)` and `x.len()` compile to the same code; this book uses the method form (`x.len()`) consistently — it reads better when chained with other methods and matches the idiom for stdlib collections.

0.7 1.5 A First Real Program: The Calculator

Now let's build something. We'll write a calculator that evaluates arithmetic expressions. This program will grow through the next several chapters — each chapter adds the next layer (types, error handling, classes, generics) as we need it.

Start with the simplest thing that works: a function that takes two numbers and an operator. Save this as `calc.ly`:

```
func eval_simple(a: f64, op: string, b: f64) -> f64 {
    if op == "+" {
        return a + b
    }
    if op == "-" {
        return a - b
    }
    if op == "*" {
        return a * b
    }
    if op == "/" {
        return a / b
    }
    return 0.0
}
```

```
func main() {
    println(f"2 + 3 = {eval_simple(2.0, "+", 3.0)}")
    println(f"10 - 4 = {eval_simple(10.0, "-", 4.0)}")
    println(f"6 * 7 = {eval_simple(6.0, "*", 7.0)}")
    println(f"15 / 3 = {eval_simple(15.0, "/", 3.0)}")
}
```

Compile and run:

```
$ lyric compile calc.ly
$ ./calc
2 + 3 = 5
10 - 4 = 6
6 * 7 = 42
15 / 3 = 5
```

(Remember from §1.4: Lyric formats `f64` with `%g`, so `5.0` prints as `5`.)

This works, but it has obvious gaps. `op` is a `string`, so the compiler can't tell `"+"` apart from `"plus"` or `"mod"` — typos slip through silently and `eval_simple` returns `0.0`. Division by zero will crash — we'll fix that in Chapter 5. And it can't handle expressions like `3 + 4 * 2` where operator precedence matters: for that we need types that can represent tokens, distinguish operators from numbers, and match on them exhaustively. That's Chapter 2.

0.8 Chapter 2: Types That Fit the Problem

The calculator from Chapter 1 takes two numbers and an operator string. It works, but it's fragile — pass `"mod"` as the operator and you silently get `0.0`. We need types that make invalid states unrepresentable.

0.9 2.1 Structs

A struct is a named group of fields:

```
struct Point {
    x: i32
    y: i32
}
```

```
func main() {
    let p = Point { x: 10, y: 20 }
    println(f"{p.x},{p.y}")
}
```

Output: `10,20`

Fields are accessed with dot notation. You can also construct structs positionally when the meaning is obvious — but only in contexts where the parser can distinguish a struct literal from a code block. Those contexts are: inside parentheses, inside function arguments, and inside list literals:

```
let pair = (Point { 10, 20 }, "origin")    // inside parens - ok
let pts = [Point { 1, 2 }, Point { 3, 4 }] // inside a list - ok
draw(Point { 0, 0 })                       // function argument -

let p = Point { x: 10, y: 20 }             // bare let: named form
```

A bare `let p = Point { 10, 20 }` is rejected — at statement level the `{` is ambiguous with a block, so you must name the fields.

Structs are value types. This is the single most important thing to understand about Lyric's type system. When you assign a struct, you copy it:

```
let p1 = Point { x: 1, y: 2 }
let mut p2 = p1     // p2 is a COPY of p1
p2.x = 99
println(f"{p1.x}")  // prints 1, not 99
```

If you come from Go, think of structs as plain `struct` values, not pointers. If you come from Rust, same — `Copy` by default, always. This will bite you exactly once, when you modify a struct and wonder why the original didn't change. After that, you'll remember.

Structs can nest:

```
struct Rect {
    top_left: Point
    bottom_right: Point
}
```

```
let r = Rect {
    top_left: Point { x: 0, y: 0 },
    bottom_right: Point { x: 100, y: 200 }
}
println(r.bottom_right.y)  // 200
```

0.10 2.2 Enums

Now back to the calculator. The operator problem — `"+"`, `"-"`, `"*"`, `"/"` as strings with no compiler checks — is exactly what enums solve.

A simple enum is a set of named constants:

```
enum Color { Red Green Blue }
```

No values, no payloads, just names. You use them directly:

```
let c = Red
```

Variants are normally unqualified — `Red`, not `Color.Red`. If two enums in the same scope share a variant name, you disambiguate by qualifying: `Color.Red` vs `Traffic.Red`. Both forms work in expressions and in `match` patterns; the qualified form is only required when bare resolution would be ambiguous.

For the calculator, here's what we actually want:

```
enum Op { Add Sub Mul Div }

func eval(a: f64, op: Op, b: f64) -> f64 {
    return match op {
        Add => { a + b }
        Sub => { a - b }
        Mul => { a * b }
        Div => { a / b }
    }
}
```

No more `"mod"` slipping through. If someone adds a `Pow` variant to `Op`, the compiler will flag every `match` that doesn't handle it.

0.11 2.3 Match

`match` is exhaustive — the compiler requires you to handle every variant. It works as an expression (returns a value) or as a statement.

Branch-type unification for `match`-as-expression is not enforced yet — the checker takes the type of the first arm and trusts the rest agree. Mixing types across arms compiles today and fails downstream. Treat the spec rule "all arms produce the same type" as load-bearing.

```
// Expression - returns a value
let name = match c {
    Red => { "red" }
    Green => { "green" }
    Blue => { "blue" }
}

// Statement - executes side effects
match c {
    Red => { println("red") }
    Green => { println("green") }
    Blue => { println("blue") }
}
```

The wildcard `_` matches anything you haven't listed:

```
match c {
    Red => { println("stop") }
    _ => { println("go") }
}
```

Multiple patterns can share an arm with `|`:

```
match c {
    Red | Blue => { println("primary") }
    Green => { println("secondary") }
}
```

0.12 2.4 Enums with Payloads

Simple enums are fine for operators, but tokens in a calculator carry data — a number token has a value, an operator token has an operator. Lyric enums handle this:

```
enum Token {
    Number(value: f64)
    Operator(op: Op)
    LeftParen
    RightParen
}
```

Each variant can carry its own set of fields. `Number` holds a `f64`. `Operator` holds an `Op`. `LeftParen` and `RightParen` carry nothing.

(This `Token` design captures parsed values — good for learning pattern matching. In Chapter 4, we'll redesign it for a real tokenizer, where carrying raw source text and position information is more useful than pre-parsed values. That's normal — types evolve as requirements do.)

Construct them by name:

```
let t1 = Number(3.14)
let t2 = Operator(Add)
let t3 = LeftParen
```

Enum variant construction is **positional only** — `Number(3.14)`, never `Number(value: 3.14)`. Named-argument syntax is reserved for struct literals (`Token { kind: TokenKind.Number, text: "42" }`). The payload field names exist so that `match` patterns can bind them, but they don't appear at the construction site.

Extract data with `match`:

```
func describe(t: Token) -> string {
    return match t {
        Number(v) => { f"number: {v}" }
        Operator(op) => {
            let name = match op {
                Add => { "+" }
                Sub => { "-" }
                Mul => { "*" }
                Div => { "/" }
            }
            f"operator: {name}"
        }
        LeftParen => { "(" }
```

```
        RightParen => { ")" }
    }
}
```

The variables in the pattern (`v`, `op`) bind to the payload fields for the duration of that arm.

0.13 2.5 Nested Patterns

Patterns nest. If you have an optional shape:

```
enum Shape {
    Circle(radius: f64)
    Rect(w: f64, h: f64)
}

enum Option {
    Some(value: Shape)
    None
}
```

We're defining our own `Option` here because we haven't covered Lyric's built-in optional type yet. In Chapter 3 we'll meet `T?` — a built-in optional that subsumes this pattern for any type, no hand-rolled enum needed.

```
func describe(opt: Option) -> string {
    return match opt {
        Some(Circle(r)) => { f"circle with radius {r}" }
        Some(Rect(w, h)) => { f"rect {w}x{h}" }
        None => { "nothing" }
    }
}

func main() {
    println(describe(Some(Circle(3.14))))
    println(describe(None))
}
```

Output:

```
circle with radius 3.14
nothing
```

The compiler destructures through `Some` and into `Circle` or `Rect` in a single pattern. No intermediate variables, no casting.

0.14 2.6 Guards

Sometimes a pattern alone isn't enough — you need a condition. Guards add `if` after the pattern:

```
func classify(n: i32) -> string {
    return match n {
        x if x < 0 => { "negative" }
        0 => { "zero" }
        x if x > 100 => { "large" }
        _ => { "positive" }
    }
}
```

The variable `x` binds to the matched value, then the guard condition is checked. If the guard fails, matching continues with the next arm. The wildcard `_` at the end catches everything the guards didn't.

0.15 2.7 `if let` and `let..else`

When you only care about one variant of an enum, a full `match` is ceremony. `if let` handles this:

```
func get_radius(s: Shape) -> f64 {
    if let Circle(r) = s {
        return r
    }
    return 0.0
}
```

The inverse is `let..else` — extract or bail:

```
func get_radius(s: Shape) -> f64 {
    let Circle(r) = s else {
        return -1.0
    }
    return r
}
```

`let..else` is particularly useful when the non-matching case is the

early return. The variable `r` is available after the `let..else` statement, in the normal flow of the function.

`if let` works with any pattern `match` accepts — variant patterns, nested patterns, all of it. `let..else` is narrower: today the pattern must be a variant pattern that starts with an uppercase identifier followed by `(...)` (e.g. `let Circle(r) = s else { ... }`). Use `if let` when you want to do something specific with one variant. Use `let..else` when you want to bail early if the variant doesn't match.

The else block is required to diverge (return, break, or panic), but the checker doesn't enforce that yet — divergence is convention today.

0.16 2.8 The `is` Operator

Sometimes you just need to know what variant you have, without extracting anything:

```
let s = Circle(3.14)
if s is Circle {
    println("it's a circle")
}
```

`is` returns a `bool`. It's the right tool when you need a type check in a condition but don't need the payload.

```
enum Shape {
    Circle(radius: f64)
    Rectangle(width: f64, height: f64)
    Point
}

func describe(s: Shape) -> string {
    if s is Circle {
        return "circle"
    }
    if s is Rectangle {
        return "rectangle"
    }
    return "point"
}
```

0.17 2.9 The as Cast

Lyric widens numeric types implicitly — `i32` to `i64` just works. Everything else requires `as`:

```
let big: i64 = 100
let small: i32 = big as i32
```

Narrowing casts truncate silently: `i64` to `i32` wraps. Float-to-integer casts truncate toward zero. These are the C rules — if you need range checking, write a function.

The checker is permissive about `as` today. Any type-to-type cast is accepted; the cast simply re-tags the value with the target type and the C backend deals with what comes out. The spec intent is to restrict it to numeric numeric (checked) and class class (checked), rejecting nonsense like `"hello" as Point`. Until that tightening lands, treat `as` as a discipline tool: use it only where the operation makes physical sense.

Casts compose in expressions:

```
let a: i32 = 10
let b: i64 = (a as i64) + (20 as i64)
```

You'll need `as` for narrowing and cross-type conversions. Widening is implicit; everything else is explicit.

0.18 2.10 The Calculator with Real Types

Now let's rewrite the calculator from Chapter 1 with proper types. Instead of passing strings as operators, we use enums:

```
enum Op { Add Sub Mul Div }

func eval(a: f64, op: Op, b: f64) -> f64 {
    return match op {
        Add => { a + b }
        Sub => { a - b }
        Mul => { a * b }
        Div => { a / b }
    }
}
```

```
func op_to_string(op: Op) -> string {
    return match op {
        Add => { "+" }
        Sub => { "-" }
        Mul => { "*" }
        Div => { "/" }
    }
}

func main() {
    let a = 2.0
    let b = 3.0
    let ops = [Add, Sub, Mul, Div]
    for op in ops {
        let result = eval(a, op, b)
        let sym = op_to_string(op)
        println(f"{a} {sym} {b} = {result}")
    }
}
```

Output:

```
2 + 3 = 5
2 - 3 = -1
2 * 3 = 6
2 / 3 = 0.666667
```

The improvement over Chapter 1 is structural. If we add a `Mod` variant to `Op`, the compiler forces us to handle it in both `eval` and `op_to_string`. String-based dispatch can't do that. Division by zero still crashes — we'll fix that in Chapter 5.

We've also introduced the `Token` type that a real parser would produce. But parsing a string like `"3 + 4 * 2"` into tokens, and evaluating those tokens with correct precedence, requires more machinery — classes for the evaluator's state, optionals for "maybe there's no more input," and methods on types. That's Chapter 3.

0.19 Chapter 3: Classes and Functions

At the end of Chapter 2, we had a calculator that uses enums for operators and structs for tokens. But we evaluated expressions by

calling `eval(a, op, b)` — one operation at a time, no memory, no state. A real expression evaluator needs to accumulate values, track pending operators, and decide when to apply them. It needs *state*.

In Lyric, that means classes.

0.20 3.1 A Class for State

Here's a stack-based calculator evaluator. Read the code first:

```
class ExprEval {
    values: [f64]
    ops: [Op]

    func push_value(self, v: f64) {
        self.values.push(v)
    }

    func push_op(self, op: Op) {
        self.ops.push(op)
    }

    func pop_value(self) -> f64? {
        if self.values.len() == 0 {
            return null
        }
        return self.values.pop()
    }

    func pop_op(self) -> Op? {
        if self.ops.len() == 0 {
            return null
        }
        return self.ops.pop()
    }

    func apply_top(self) -> bool {
        let op = self.pop_op()
        if isnull(op) {
            return false
        }
```

```
        let b = self.pop_value()
        let a = self.pop_value()
        if isnull(a) || isnull(b) {
            return false
        }
        let result = eval(a!, op!, b!)
        self.push_value(result)
        return true
    }

    func result(self) -> f64? {
        if self.values.len() == 0 {
            return null
        }
        return self.values[0]
    }
}
```

Several things are new here. Let's take them in order.

Classes are heap-allocated. When you write `let ev = ExprEval {}`, Lyric allocates the object on the heap and `ev` holds a reference to it. This is the fundamental difference from structs: structs are values that get copied on assignment, classes are references that get shared. If you pass `ev` to a function, both the caller and the function see the same object.

Methods take `self`. A method declared inside a class body receives the instance as `self`. Since classes are references, `self` is always mutable — you can assign to `self.values` without any special annotation. (Structs are different, as we'll see in §3.7.)

The `?` in return types. `pop_value` returns `f64?` — a value that might be `null`. This is Lyric's optional type, and it's how the evaluator handles the case where you try to pop from an empty stack. We'll cover optionals properly in §3.4.

0.21 3.2 Using the Evaluator

```
func main() {
    // Evaluate: 3 + 4 * 2
    let ev = ExprEval {}
```

```
    ev.push_value(3.0)
    ev.push_op(Add)
    ev.push_value(4.0)
    ev.push_op(Mul)
    ev.push_value(2.0)

    // Apply * first (higher precedence)
    ev.apply_top()
    // Apply +
    ev.apply_top()

    let r = ev.result()
    if !isnull(r) {
        println(f"3 + 4 * 2 = {r!}")
    }
}
```

Output:

```
3 + 4 * 2 = 11
```

The evaluator manages the precedence dance manually here — we push all values and operators, then apply `*` before `+`. This uses the `Op` enum and `eval` function from Chapter 2. A proper recursive-descent parser would handle precedence automatically. We're building toward that, but the point right now is that `ExprEval` holds state across multiple calls. That's what classes are for.

Notice the construction syntax: `ExprEval {}`. Class constructors use the same curly-brace syntax as struct literals, but since `values` and `ops` default to empty slices, we don't need to specify them. You could also write `ExprEval { values: [], ops: [] }` — same thing.

0.22 3.3 Classes vs Structs

This distinction matters and will keep coming back:

	Struct	Class
Allocated	Stack (value)	Heap (reference)
Assignment	Copies the data	Copies the reference

	Struct	Class
Identity	None — two copies are independent	Two references can point to the same object
Passed to functions	By value (copied)	By reference (shared)

The `Token` enum from Chapter 2 is the right choice for tokens — each token is a small immutable variant (a number with its `f64` payload, an operator with its `Op` payload, or a paren). `ExprEval` is the right choice for the evaluator — it has identity (there's *this* evaluator), it mutates, and you want functions to see the same object.

The rule of thumb: if it's data, use a struct. If it's a thing with behavior and identity, use a class.

0.23 3.4 Optionals

`pop_value` returns `f64?` — an `f64` that might be absent. The `?` suffix makes any type optional. An optional value is either the underlying type or `null`:

```
func find(xs: [i32], target: i32) -> i32? {
    for x in xs {
        if x == target {
            return x
        }
    }
    return null
}

func main() {
    let xs = [10, 20, 30, 40, 50]

    let found = find(xs, 30)
    if !isnull(found) {
        println(f"found: {found!}")
    }

    let missing = find(xs, 99)
```

```
    if isnull(missing) {
        println("not found: correct")
    }

    println(f"direct unwrap: {find(xs, 20)!}")
}
```

Output:

```
found: 30
not found: correct
direct unwrap: 20
```

Three operations on optionals:

- `isnull(x)` — returns `true` if `x` is `null`
- `x!` — unwraps the value, crashes if `null` (the "I know it's there" operator)
- `null` — the absent value

You might wonder why we use `isnull(x)` + `x!` instead of the `match` from Chapter 2. Both work. Use `match` when you need to destructure or bind the inner value to a new name. Use `isnull`/`!` for simple presence checks — it's more concise and the idiomatic choice for most Lyric code.

The `!` operator is a deliberate trade-off. It's concise for cases where you've already checked, and it crashes loudly when you're wrong. No silent null propagation, no billion-dollar mistake — you either check or you crash.

Optional types compose: `string?` is an optional string, `[i32]?` is an optional slice. You can't accidentally use an optional where a concrete type is expected — the compiler forces you to unwrap first.

0.23.1 Auto-Deref for Optional Class Receivers

There's one place where Lyric does NOT force you to write `!`: field access on an optional whose inner type is a **class**. The checker auto-unwraps:

```
class Node {
    name: string
    next: Node?
```

```
}

func greet(n: Node?) {
    println(n.name)            // n is Node?, .name accessed direc
    if !isnull(n.next) {
        println(n.next.name)   // chained auto-deref also works
    }
}
```

The convenience pays for itself in linked-list and AST traversal code, where every link is "guaranteed non-null in this branch" and `!` markers become noise. It applies only to class optionals — struct and primitive optionals (`Point?`, `i32?`) still require explicit `!` because they use a tagged representation under the hood.

Today, if the optional actually is null when accessed this way, the C backend segfaults — the lowerer emits a direct field load with no runtime null check. That's a bug, not a feature. The fix on the roadmap is to emit the same Lyric-level panic that ***`expr!`*** *produces. Until then: if your control flow doesn't already prove the value is non-null, write* ***`n!.name`*** *for an honest panic, or guard with* ***`if !isnull(n) { ... }`****.*

0.23.2 Lvalue Unwrap — Writing Through `!`

`expr!` isn't just an rvalue; it's also a valid lvalue. When the inner type is a class, you can write through the unwrap to mutate a field on the unwrapped object in place:

```
class Outer { data: Inner? }
class Inner { value: i32 }

let o = Outer { data: Inner { value: 0 } }
o.data!.value = 42        // writes through to the Inner object
println(o.data!.value)    // 42
```

This is the right idiom whenever you have a "this is initialized once, mutated many times" field. The unwrap panics on null exactly as in the rvalue case.

Lvalue write-through is reliable today only when the inner type is a ***`class`*** *(a heap reference, as above). If* ***`Inner`*** *is a struct or primitive,*

the optional uses a tagged representation, and `o.data!.value = 42` silently writes to a temporary copy — the change is lost. Until that's fixed, model "mutable inner state" as a class, or pull the struct out, mutate it, and assign it back: `let mut tmp = o.data!; tmp.value = 42; o.data = tmp`.

0.24 3.5 Methods Inside and Outside

So far we've defined methods inside the class body. Lyric also lets you define methods externally:

```
class Counter {
    count: i32

    func increment(self) {
        self.count = self.count + 1
    }

    func get(self) -> i32 {
        return self.count
    }
}

// External method - defined outside the class body
func Counter.reset(self) {
    self.count = 0
}
```

Both forms call the same way: `c.increment()`, `c.reset()`. External methods exist for a specific reason: they let interfaces add methods to types without modifying the type's source file. We'll use this extensively in Chapter 9 when we build multi-class interfaces. For now, just know it's there.

0.25 3.6 Visibility

By default, fields and methods are private to the package. Add `pub` to export them:

```
class Counter {
    count: i32            // private - only this package can acces
```

```
    pub func increment(self) {   // public - any importer can call
        self.count = self.count + 1
    }

    pub func get(self) -> i32 {  // public
        return self.count
    }
}
```

Lyric's default is private because most fields are implementation details. You export the interface, not the internals.

A note on naming. Lyric's compiler is case-agnostic — there is no Go-style "capital means exported" rule (that's what `pub` is for). The conventions below are convention only, but the ecosystem follows them and your code will read better if you do too:

Kind	Convention	Example
Classes, structs, enums, interfaces	PascalCase	`Counter`, `Point`, `Color`, `Graph`
Enum variants	PascalCase	`Red`, `Circle`, `LeftParen`
Type variables	Short PascalCase	`T`, `U`, `P`, `C`
Functions and methods	snake_case	`array_append`, `get_hash`
Fields	snake_case	`roster_children`, `is_empty`
Locals and parameters	snake_case	`let total_count = 0`
Module-level constants	UPPER_SNAKE	`let PREC_NONE: i32 = 0`
Packages	snake_case	`ast`, `parser`, `expr_parser`
Test functions	`test_` prefix	`test_lexer_basic`

The `test_` prefix is the one rule the compiler does enforce — the test runner discovers tests by it (Chapter 7). Everything else is style.

One catch the compiler enforces ruthlessly: **field-literal construction must match the declared name exactly.** `Point { x: 1.0`

`}` works because the field is `x`. `Point { X: 1.0 }` is a checker error. No case-insensitive matching, no fuzzy resolution, no automatic PascalCase snake_case translation. If you mis-case a field name, you get a clear error at the construction site.

0.26 3.7 `mut` Parameters — When Structs Need to Change

Classes are always passed by reference — mutations are visible to the caller. Structs are different. Since structs are values, passing one to a function copies it. If you want a function to modify a struct in place, you need `mut`:

```
struct Point {
    x: i32
    y: i32
}

func translate(mut p: Point, dx: i32, dy: i32) {
    p.x = p.x + dx
    p.y = p.y + dy
}

func main() {
    let mut p = Point { x: 10, y: 20 }
    translate(mut p, 5, 3)
    println(f"({p.x}, {p.y})")
}
```

Output:

```
(15, 23)
```

`mut` appears in three places: the parameter declaration (`mut p: Point`), the call site (`translate(mut p, ...)`), and the variable declaration (`let mut p`). All three are required. This is deliberate — when you read a call site and see `mut`, you know that argument might be modified. No surprises.

Why not just use a class? Because Point is data — two integers, no identity, no heap allocation needed. `mut` gives you pass-by-reference for value types when you need it, without forcing everything onto

the heap.

One small point about class methods: you may see `mut self` written in older code or in code translated from Rust. The parser accepts it, but the `mut` is redundant — `self` on a class method is always mutable (classes are reference types, so the method already operates through a pointer). Prefer plain `self`.

0.27 3.8 Lambdas and Higher-Order Functions

Lyric supports two lambda syntaxes. The pipe style:

```
let double = |x: i32| -> i32 { x * 2 }
```

And the paren style:

```
let double = (x: i32) -> i32 { x * 2 }
```

Both work identically. Pipe style is conventional for short lambdas; paren style reads better when the parameter list is complex.

Lambdas are values. You can pass them to functions:

```
func apply(x: i32, f: func(i32) -> i32) -> i32 {
    return f(x)
}

func main() {
    let result = apply(7, |x: i32| -> i32 { x + 3 })
    println(result)
}
```

Output:

```
10
```

The type `func(i32) -> i32` is a function type — any function or lambda matching that signature. We could use this to make `eval` more flexible by letting the caller plug in operations beyond the four `Op` variants. In Chapter 6, we'll see lambdas compose with generic functions to build reusable higher-order operations like `transform` and `filter`.

0.28 3.9 A Proper Stack

Our `ExprEval` has `values` and `ops` as raw slices with manual push/pop logic. Let's extract a reusable stack:

```
class Stack {
    items: [f64]

    func push(self, item: f64) {
        self.items.push(item)
    }

    func pop(self) -> f64? {
        if self.items.len() == 0 {
            return null
        }
        return self.items.pop()
    }

    func size(self) -> i32 {
        return self.items.len()
    }
}
```

This is the same pop-with-optional pattern from `ExprEval`, but now it's a standalone class. We could rewrite `ExprEval` to use two `Stack` instances instead of managing slices directly.

This stack only holds `f64` values. If we wanted a stack of strings, we'd have to write a second class with identical logic. That duplication is exactly what generics solve — in Chapter 6, we'll make `Stack<T>` work for any type. For now, the concrete version does what the calculator needs.

0.29 3.10 The Calculator So Far

The evaluator is a class because it holds state — two stacks that grow and shrink across method calls. What's still missing: we're feeding values and operators by hand. A real calculator takes a string like `"(5 + 3) * 2"` and produces tokens automatically. That requires string indexing, character-by-character scanning, and slices — Chapter 4.

A Glimpse Ahead: Relations

Our calculator's `Expr` nodes will eventually form trees — parents pointing to children, children needing cleanup when parents are destroyed. In most languages, you'd write that ownership logic by hand (C++), fight a borrow checker for it (Rust), or accept garbage collection pauses (Go). In Lyric, you'll write one line:

```
relation ArrayList Expr:children owns [Expr:parent]
```

The compiler generates the child array, parent back-pointer, cascade destructors, and removal logic. No runtime cost, no annotation burden. That's Chapter 8 — and it's the feature that makes Lyric different from everything else.

0.30 Chapter 4: Strings, Slices, and Collections

We ended Chapter 3 with a calculator that evaluates expressions — but only when we feed it values and operators by hand. A real calculator takes a string like `"(5 + 3) * 2"` and figures out what to do with it. That means scanning text character by character, which means we need to understand how Lyric handles strings.

0.31 4.1 Strings Are Byte Slices

In Lyric, `string` is represented as `[u8]` — a slice of bytes. The type keeps its own name (`string` everywhere in your code), but everything you learn about slices in this chapter applies to strings, and the string-specific methods are a thin layer over byte operations.

```
func main() {
    let s = "Hello"
    println(f"length: {s.len()}")    // 5
    println(f"first byte: {s[0]}")   // 72 (ASCII 'H')
    println(f"last byte: {s[4]}")    // 111 (ASCII 'o')
}
```

Indexing a string returns a `u8`, not a character. There is no character type — `u8` serves that role. Character literals like `'A'` produce `u8` values:

```
func main() {
    let a: u8 = 'A'
    let z: u8 = 'Z'
    println(f"A = {a}")   // A = 65
    println(f"Z = {z}")   // Z = 90

    let nl: u8 = '\n'     // newline
    let tb: u8 = '\t'     // tab
    let hex: u8 = '\x41'  // hex literal - also 65, also 'A'
}
```

This is the same model as C and Go: a string is a sequence of bytes, not Unicode code points. `s.len()` is the **byte** length; `s[i]` is the byte at offset `i`. ASCII text works perfectly. UTF-8 text round-trips through I/O and concatenation just fine, but iterating "characters" means iterating bytes — a multi-byte code point shows up as several consecutive `u8` values.

Roadmap: a UTF-8 layer is planned — `\u{NNNN}` escapes, code-point iteration (`for c in s.chars()`), `char_at` returning an `i32` code point, Unicode-aware case operations. The `string` type name stays; the byte-level operations stay; the new operations layer on top. Until then, treat `string` as `[u8]` and write Unicode-aware code by hand.

For the calculator we're building — and for compilers, network protocols, and most systems code — bytes are exactly what you want.

0.32 4.2 String Methods

Strings come with the methods you'd expect. Here are the ones we'll use in the tokenizer:

```
func main() {
    let s = "hello, world"
    println(f"length: {s.len()}")                // 12
    println(f"contains: {s.contains("world")}")  // true
    println(f"index_of: {s.index_of("world")}")  // 7
```

```
    println(f"trim: '{"  hi  ".trim()}'")        // 'hi'

    let csv = "a,b,c,d"
    let parts = csv.split(",")
    println(f"parts: {parts.len()}")             // 4
    println(f"rejoin: {parts.join(" | ")}")      // a | b | c | d
}
```

`.index_of()` returns the byte offset, or -1 if not found — the C convention, not an optional. For a method you typically use in comparisons (`if s.index_of("x") >= 0`), the sentinel is cleaner than unwrapping. `.split()` returns `[string]` — a slice of strings.

Lyric also provides `.replace()`, `.repeat()`, `.has_prefix()`, `.has_suffix()`, `.to_upper()`, `.to_lower()` — they work as you'd expect, and we'll use them when we need them.

0.33 4.3 Slices

A slice `[T]` is a fat pointer: data, length, and capacity. Slices are Lyric's general-purpose dynamic array.

```
func main() {
    let mut items: [i32] = []
    items.push(10)
    items.push(20)
    items.push(30)
    println(f"length: {items.len()}")       // 3
    println(f"contains 20: {items.contains(20)}")  // true

    let last = items.pop()
    println(f"popped: {last}")              // 30
    println(f"after pop: {items.len()}")    // 2
}
```

`.push()` appends to the end. `.pop()` removes and returns the last element. `.contains()` does a linear search. These are the same methods we used on the `Stack` class in Chapter 3, because `Stack.items` was a `[f64]` underneath.

Slices support concatenation with `+`:

```
func main() {
```

```
    let a = [1, 2, 3]
    let b = [4, 5]
    let c = a + b
    println(f"length: {c.len()}")   // 5
    println(f"first: {c[0]}")       // 1
    println(f"last: {c[4]}")        // 5

    // originals are unchanged
    println(f"a still: {a.len()}")  // 3
}
```

The `+` operator creates a new slice. The originals are unmodified. For in-place growth, push elements one at a time, or use the `append` built-in:

```
func main() {
    let mut xs = [1, 2, 3]
    let more = [4, 5, 6]
    let mut i = 0
    while i < more.len() {
        xs.push(more[i])
        i = i + 1
    }
    println(f"length: {xs.len()}")  // 6

    // Or use the append built-in (returns a new slice - re-bind):
    let mut ys = [1, 2, 3]
    ys = append(ys, 4)
    ys = append(ys, 5)
    println(f"length: {ys.len()}")  // 5
}
```

A method `xs.extend(ys)` is listed in the spec as the canonical in-place append-all, but today it's a silent no-op — the slice's length doesn't change. Until that's wired up, use `push` in a loop or the `append` built-in.

Slice expressions extract a sub-range:

```
func main() {
    let s = "hello, world"
    let hello = s[0:5]
```

```
    let world = s[7:12]
    println(hello)   // hello
    println(world)   // world
}
```

`s[lo:hi]` returns elements from index `lo` up to but not including `hi`. This works on any slice, not just strings.

Three shorthand forms drop one or both endpoints:

```
let s = "hello, world"
let head = s[:5]      // same as s[0:5]   → "hello"
let tail = s[7:]      // same as s[7:s.len()]  → "world"
let copy = s[:]       // full descriptor copy (shares backing arra
```

`xs[:n]` defaults the low end to 0, `xs[n:]` defaults the high end to the slice length, and `xs[:]` does both. The last form is the idiomatic way to take a fresh slice descriptor that shares the same backing array — useful when you want to hand a slice to a function without letting its `push` operations resize your local view.

0.33.1 Copy semantics and `let ref`

Slices are value types, but the value is just the descriptor — a pointer, a length, and a capacity. Assignment copies the descriptor; the backing array is shared:

```
let a = [10, 20, 30]
let b = a             // copies the descriptor; b and a point to th
println(f"{b[0]} {b[1]} {b[2]}")   // 10 20 30
```

A `let ref` binding makes the sharing explicit and skips even the descriptor copy:

```
let buf = "hello, world"
let ref head = buf[0:5]    // zero-copy view into buf
println(head)              // hello
```

`let ref` is the right binding when you're walking a buffer and want a name for "this slice of those bytes" without paying for any copy. The source must outlive the `ref`. Plain `let` is fine for almost everything; reach for `ref` when you're writing a parser, a serializer, or any hot loop that takes sub-views of a buffer thousands of times.

0.34 4.4 Scanning Text

Now we have the tools to build a tokenizer. In Chapter 2, we defined `Token` as an enum with payloads — `Number(value: f64)`, `Operator(op: Op)`, `LeftParen`, `RightParen`. That design was right for learning pattern matching, but a real tokenizer needs something different: the raw text of each token, not a pre-parsed value. Parsing `"3.14"` into `f64` is the parser's job, not the lexer's. We also want to be able to add source position later (line and column, for error messages) without rewriting every variant.

So we redesign in two pieces: a flat `TokenKind` enum that just names the *kind* of token, and a `Token` struct that carries the kind plus the source text the lexer saw. The paren variants keep their full names — `LeftParen` and `RightParen` — to match Chapter 2 and stay readable in `match` arms:

```
enum TokenKind {
    Number
    Plus
    Minus
    Star
    Slash
    LeftParen
    RightParen
}

struct Token {
    kind: TokenKind
    text: string
}
```

Why the split? The enum-with-payloads form from Chapter 2 conflates two things: classifying the token (it's a number) and holding the data the parser needs (the value 3.14). A real lexer wants only the first — let the parser convert text to numbers, and let the lexer focus on slicing the input. The struct also gives us a single, stable place to add fields later (`line: i32`, `col: i32`, source-file index) without disturbing the seven variants.

The interesting part of the tokenizer is scanning multi-character tokens. Single characters like `+` and `(` are trivial — one byte comparison,

one token. Numbers require a loop:

```
// Inside the tokenizer loop, when ch >= '0' && ch <= '9':
let start = pos
while pos < input.len() && input[pos] >= '0' && input[pos] <= '9'
    pos = pos + 1
}
// Handle decimal point
if pos < input.len() && input[pos] == '.' {
    pos = pos + 1
    while pos < input.len() && input[pos] >= '0' && input[pos] <=
        pos = pos + 1
    }
}
tokens.push(Token { kind: TokenKind.Number, text: input[start:pos]
```

`input[start:pos]` slices out the number's text — `"3"`, `"42"`, `"3.14"`. Because strings are byte slices, this is a descriptor copy, not a string allocation. The byte comparisons `ch >= '0' && ch <= '9'` are the same digit check you'd write in C. Character literals make the intent readable: `input[pos] == '.'` instead of `input[pos] == 46`.

To include literal braces in f-string output, double them:

```
println(f"token {{kind}}: {tok.text}")
// prints: token {kind}: 42
```

0.35 4.5 StringBuilder

String concatenation with `+` creates a new string each time. For building strings in a loop, that's $O(n^2)$. `StringBuilder` gives you $O(n)$:

```
func main() {
    let sb = new_string_builder()  // stdlib constructor function
    sb.write("hello")
    sb.write(" ")
    sb.write("world")
    println(sb.to_string())        // hello world
    println(f"{sb.len()}")         // 11
}
```

`StringBuilder` is a class — it's heap-allocated and mutated through method calls. `.write()` appends a string. `.write_byte()` appends a single `u8`. `.to_string()` produces the final result.

For strings with embedded quotes, triple-quote syntax avoids escaping:

```
let json = """{"name": "Lyric", "version": 1}"""
println(json)
// prints: {"name": "Lyric", "version": 1}
```

0.36 4.6 Tuples

Tuples are anonymous structs with positional fields. They're useful for returning multiple values:

```
func make_pair() -> (i32, string) {
    return (10, "hello")
}

func main() {
    let p = make_pair()
    println(p._0)    // 10
    println(p._1)    // hello
}
```

Fields are accessed with `._0`, `._1`, `._2`, and so on — tuples can have any number of elements. You can also destructure:

```
func main() {
    let (val, ok) = atoi("99")
    if ok {
        println(f"parsed: {val}")   // parsed: 99
    }
}
```

We already saw this pattern with `atoi()`, which returns `(i64, bool)` — the parsed integer and whether parsing succeeded. (Lyric's default integer literal type is `i32`, so the `99` you wrote in the input string becomes the `i64` value `99` here; cast with `val as i32` if you need it narrower.) Tuples and destructuring eliminate the need for out-parameters or wrapper structs when a function returns two things.

0.37 4.7 Conversion Functions

Three built-in functions handle the most common conversions:

```
func main() {
    // int → string
    let s = itoa(42)
    println(s)                      // 42
    println(itoa(-123))             // -123

    // string → int
    let (val, ok) = atoi("99")
    if ok {
        println(val)                // 99
    }

    let (_, ok2) = atoi("not_a_number")
    if !ok2 {
        println("parse failed")   // parse failed
    }

    // byte → string
    let c: u8 = 'A'
    let cs = char_to_string(c)
    println(cs)                     // A
}
```

`atoi` returns `(i64, bool)` — the parsed value and whether parsing succeeded. No exceptions, no error types. The `(T, bool)` pattern is Go-influenced; you could also use `i64?`, but for simple conversions the bool convention keeps call sites flat. The companion `parse_float(s) -> (f64, bool)` does the same job for floating-point — that's the one the parser uses to turn a `Number` token's text into the numeric value it evaluates. We'll see proper error handling in Chapter 5.

0.38 4.8 The Complete Tokenizer

Here's the complete tokenizer using the `TokenKind` and `Token` types from §4.4:

```
enum TokenKind {
    Number
    Plus
    Minus
    Star
    Slash
    LeftParen
    RightParen
}

struct Token {
    kind: TokenKind
    text: string
}

func tokenize(input: string) -> [Token] {
    let mut tokens: [Token] = []
    let mut pos = 0

    while pos < input.len() {
        let ch = input[pos]

        if ch == ' ' || ch == '\t' || ch == '\n' {
            pos = pos + 1
            continue
        }

        if ch == '(' {
            tokens.push(Token { kind: TokenKind.LeftParen, text: "
            pos = pos + 1
        } else if ch == ')' {
            tokens.push(Token { kind: TokenKind.RightParen, text:
            pos = pos + 1
        } else if ch == '+' {
            tokens.push(Token { kind: TokenKind.Plus, text: "+" })
            pos = pos + 1
        } else if ch == '-' {
            tokens.push(Token { kind: TokenKind.Minus, text: "-" }
            pos = pos + 1
        } else if ch == '*' {
```

```
            tokens.push(Token { kind: TokenKind.Star, text: "*" })
            pos = pos + 1
        } else if ch == '/' {
            tokens.push(Token { kind: TokenKind.Slash, text: "/" }
            pos = pos + 1
        } else if ch >= '0' && ch <= '9' {
            let start = pos
            while pos < input.len() && input[pos] >= '0' && input[
                pos = pos + 1
            }
            if pos < input.len() && input[pos] == '.' {
                pos = pos + 1
                while pos < input.len() && input[pos] >= '0' && in
                    pos = pos + 1
                }
            }
            tokens.push(Token { kind: TokenKind.Number, text: inpu
        } else {
            pos = pos + 1  // skip unknown characters - we'll add
        }
    }

    return tokens
}

func main() {
    let input = "(5 + 3) * 2"
    let tokens = tokenize(input)
    for tok in tokens {
        println(f"{tok.kind}: {tok.text}")
    }
}
```

Output:

```
LeftParen: (
Number: 5
Plus: +
Number: 3
RightParen: )
Star: *
```

```
Number: 2
```

The tokenizer uses everything from this chapter: byte indexing (`input[pos]`), character literals (`'0'`, `'9'`, `'.'`), slice expressions (`input[start:pos]`), `.push()` on a slice, and `.len()` for bounds checking. The `for tok in tokens` loop is the idiomatic iteration form — `tokens` is a `[Token]` slice, and `for ... in` walks it without a manual index.

When we push a `Token` into the slice, a copy goes in — structs are value types. The tokenizer allocates no new strings for operators (those are literals), and only creates slice descriptors for numbers — the number's text is a view into the original `input`, not a fresh allocation.

0.39 4.9 The Calculator So Far

We now have types (Chapter 2), an evaluator (Chapter 3), and a tokenizer (this chapter) that scans strings into token arrays. What's missing is the glue: a parser that reads tokens, drives evaluation, and handles malformed input. That's Chapter 5.

0.40 Chapter 5: Error Handling

The calculator tokenizes input and evaluates expressions, but it has a gap: what happens when the input is wrong? Feed `"5 + "` to the tokenizer and it produces tokens happily. Feed `"(5 + )"` and the evaluator will crash. We need a way for functions to say "this failed" without crashing.

0.41 5.1 Errors Are Values

Lyric handles errors the same way Go does: functions return them. An error is an interface — any class with a `message(self) -> string` method satisfies it. The stdlib provides a concrete `Error` class for the common case:

```
func divide(a: i32, b: i32) -> (i32, error) {
    if b == 0 {
        return (0, Error { msg: "division by zero" })
    }
```

```
    return (a / b, null)
}
```

The return type `(i32, error)` is a tuple: the result and an error. On success, the error is `null`. On failure, you return whatever value makes sense for the result (usually zero) and an error with a message. The caller checks:

```
func main() {
    let (val, err) = divide(10, 2)
    if err != null {
        println(f"Error: {err}")
    } else {
        println(f"10 / 2 = {val}")
    }

    let (_, err2) = divide(10, 0)
    if err2 != null {
        println(f"Expected error: {err2}")
    }
}
```

Output:

```
10 / 2 = 5
Expected error: division by zero
```

This is the entire error model. No exceptions, no stack unwinding, no `try`/`catch`. The error is in the return type, visible in the signature, and the caller decides what to do. If you've written Go, this is familiar. If you're coming from Rust, think of `Result<T, E>` but without needing to name the error type — `error` is always the interface.

Use `null` for the no-error case.

0.42 5.2 The ? Operator

Checking errors with `if err != null` on every call gets verbose fast. When a function just wants to propagate errors upward, the `?` operator does it in one character:

```
func compute(x: i32) -> (i32, error) {
    let result = divide(x, 2)?
```

```
    let doubled = divide(result * 4, 2)?
    return (doubled, null)
}
```

`divide(x, 2)?` calls `divide`, checks the error, and if it's non-null, immediately returns `(zero_value, err)` from `compute`. If there's no error, `?` unwraps the tuple and `result` gets just the `i32`. Without `?`, this would be:

```
func compute(x: i32) -> (i32, error) {
    let (result, err1) = divide(x, 2)
    if err1 != null {
        return (0, err1)
    }
    let (doubled, err2) = divide(result * 4, 2)
    if err2 != null {
        return (0, err2)
    }
    return (doubled, null)
}
```

The `?` version is half the code and says the same thing. The constraint: `?` only works inside functions that themselves return `(T, error)`. The compiler enforces this — you can't use `?` in `main()` unless `main` returns an error tuple.

0.43 5.3 Nested ? and Expressions

The `?` operator works inside expressions, not just in `let` statements. You can pass a fallible result directly to another function:

```
func parse_int(s: string) -> (i32, error) {
    if s == "42" {
        return (42, null)
    }
    return (0, Error { msg: f"invalid: {s}" })
}

func double(x: i32) -> i32 {
    return x * 2
}
```

```
func process(s: string) -> (i32, error) {
    let result = double(parse_int(s)?)
    return (result, null)
}
```

`parse_int(s)?` either returns the error from `process` or yields the `i32`, which flows directly into `double()`. You can also use `?` on both sides of a binary expression:

```
func add_parsed(a: string, b: string) -> (i32, error) {
    let sum = parse_int(a)? + parse_int(b)?
    return (sum, null)
}
```

If either `parse_int` fails, the error propagates. If both succeed, `sum` gets the addition of the two unwrapped values.

0.44 5.4 ? in Loops

The `?` operator works naturally in loops. Here's a function that collects items, bailing on the first failure:

```
class Item {
    name: string
}

func make_item(s: string) -> (Item, error) {
    if s == "" {
        return (Item { name: "" }, Error { msg: "empty name" })
    }
    return (Item { name: s }, null)
}

func collect(names: [string]) -> ([Item], error) {
    let mut items: [Item] = []
    let mut i = 0
    while i < names.len() {
        let item = make_item(names[i])?
        items = append(items, item)
        i = i + 1
    }
    return (items, null)
```

```
}
```

When `?` fires inside the loop, it returns from `collect`, not just from the loop iteration.

`Error { msg: "empty name" }` builds the stdlib `Error` class directly — that's the literal form we'll use throughout the rest of the chapter. The spec also lists a free-function shortcut, `new_error(msg)`, that does the same thing. *Roadmap: `new_error(msg)` type-checks today but the C backend doesn't yet emit a definition for it, so any program that calls it fails to link. Use the `Error { msg: ... }` literal until the lowering lands.*

0.45 5.5 Custom Errors

The stdlib `Error` class works for simple cases, but sometimes you want errors that carry structured data. Any class that implements a `message(self) -> string` method satisfies the `error` interface:

```
class ParseError {
    line: i32
    col: i32
    msg: string

    pub func message(self) -> string {
        return f"{self.line}:{self.col}: {self.msg}"
    }
}
```

Now `ParseError` can be returned anywhere `error` is expected:

```
func parse_token(input: string, pos: i32) -> (Token, error) {
    if pos >= input.len() {
        return (Token { kind: TokenKind.Number, text: "" },
                ParseError { line: 1, col: pos + 1, msg: "unexpect
    }
    // ... parse normally ...
}
```

The caller doesn't need to know it's a `ParseError` — it just sees `error` and can print the message. This is the same pattern as Go's `error` interface: any class with a `pub func message(self) -> string` method satisfies it.

For one-off errors where a dedicated class is overkill, the stdlib `Error` class is the right tool — that's exactly what it's there for:

```
return (0, Error { msg: "division by zero" })
```

A note on stringifying errors: `f"{err}"` automatically prints the error's message (the f-string lowerer knows about the `error` type), which is why every example so far reaches for `f"...: {err}"` rather than calling `err.message()` explicitly. *Roadmap: calling `err.message()` directly on an `error`-typed value doesn't compile today — interface dispatch for `error` isn't wired up in the C backend. The f-string form works because it has a dedicated lowering path. Concrete classes that satisfy `error` (like `ParseError`) can have `.message()` called on them directly; only the interface-typed receiver is the problem.*

0.46 5.6 A Parser for the Calculator

Now we can build the parser that connects the tokenizer to the evaluator. The parser reads `[Token]`, handles operator precedence with recursive descent, and returns errors for malformed input.

```
class Parser {
    tokens: [Token]
    pos: i32

    func peek(self) -> Token? {
        if self.pos >= self.tokens.len() {
            return null
        }
        return self.tokens[self.pos]
    }

    func next(self) -> Token? {
        let tok = self.peek()
        if tok != null {
            self.pos = self.pos + 1
        }
        return tok
    }
```

```
func expect(self, kind: TokenKind) -> (Token, error) {
    let tok = self.next()
    if tok == null {
        return (Token { kind: kind, text: "" },
                Error { msg: f"expected {kind}, got end of inp
    }
    if tok!.kind != kind {
        return (Token { kind: kind, text: "" },
                Error { msg: f"expected {kind}, got {tok!.kind
    }
    return (tok!, null)
}

// parse_primary: numbers and parenthesized sub-expressions
func parse_primary(self) -> (f64, error) {
    let tok = self.next()
    if tok == null {
        return (0.0, Error { msg: "unexpected end of input" })
    }
    if tok!.kind == TokenKind.Number {
        let val = str_to_float(tok!.text)  // stdlib: converts
        return (val, null)
    }
    if tok!.kind == TokenKind.LeftParen {
        let val = self.parse_expr()?
        self.expect(TokenKind.RightParen)?
        return (val, null)
    }
    return (0.0, Error { msg: f"unexpected token: {tok!.text}"
}

// parse_term: * and /
func parse_term(self) -> (f64, error) {
    let mut left = self.parse_primary()?
    while self.peek() != null {
        let kind = self.peek()!.kind
        if kind != TokenKind.Star && kind != TokenKind.Slash {
            break
        }
        let op = self.next()!
```

```
            let right = self.parse_primary()?
            if op.kind == TokenKind.Star {
                left = left * right
            } else {
                if right == 0.0 {
                    return (0.0, Error { msg: "division by zero" }
                }
                left = left / right
            }
        }
        return (left, null)
    }

    // parse_expr: + and -
    func parse_expr(self) -> (f64, error) {
        let mut left = self.parse_term()?
        while self.peek() != null {
            let kind = self.peek()!.kind
            if kind != TokenKind.Plus && kind != TokenKind.Minus {
                break
            }
            let op = self.next()!
            let right = self.parse_term()?
            if op.kind == TokenKind.Plus {
                left = left + right
            } else {
                left = left - right
            }
        }
        return (left, null)
    }
}

func parse(input: string) -> (f64, error) {
    let tokens = tokenize(input)
    let parser = Parser { tokens: tokens, pos: 0 }
    return parser.parse_expr()
}
```

The `?` operator makes the recursive descent clean. Every call to

`parse_primary()` or `parse_term()` can fail, and `?` propagates the error upward without cluttering the logic. Compare `let right = self.parse_primary()?` to the alternative: a three-line `let`/`if`/`return` block at every call site. The parser would be twice as long.

Notice `parse_primary` handles parenthesized sub-expressions by calling `parse_expr` recursively — mutual recursion between the precedence levels. The `?` on `self.expect(TokenKind.RightParen)?` discards the returned token (we don't need it) but propagates the error if the closing paren is missing.

A note on `tok!` after a null check: Lyric doesn't narrow optional types through control flow. After `if tok == null { return ... }`, the compiler still considers `tok` a `Token?`, so `tok!` is required. This is a deliberate simplicity tradeoff. And since `Parser` is a class (not a struct), its methods mutate `self.pos` without needing `mut` — classes are reference types, so mutation is implicit.

0.47 5.7 Putting It Together

With the parser in place, we can wire everything up:

```
func main() {
    let expressions = ["(5 + 3) * 2", "10 / 3", "1 + 2 * 3 + 4", "
    for expr in expressions {
        let (result, err) = parse(expr)
        if err != null {
            println(f"{expr} => error: {err}")
        } else {
            println(f"{expr} = {result}")
        }
    }
}
```

Output:

```
(5 + 3) * 2 = 16
10 / 3 = 3.33333
1 + 2 * 3 + 4 = 11
(5 + ) => error: unexpected token: )
```

The malformed expression `"(5 + )"` reaches `parse_primary`, which

sees `)` where it expects a number or `(`, and returns an error. The `?` in `parse_term` propagates it up through `parse_expr` and out through `parse`. No exceptions, no unwinding — just return values flowing back up the call stack.

0.48 5.8 Why Not Exceptions

Exceptions hide control flow. A `try/catch` block wrapping twenty lines of code means any of those lines might jump to the catch — you can't tell which without reading every function signature (and in most languages, not even then). Lyric's approach makes error paths visible:

- **In the signature**: `-> (f64, error)` tells you the function can fail. No surprises.
- **At the call site**: `?` marks exactly which calls can fail. Read the parser — every `?` is a potential exit point, and they're all visible.
- **Zero-cost happy path**: when there's no error, `?` is a null check and nothing more. No exception tables, no stack unwinding overhead.

The tradeoff is verbosity. In the parser, `?` keeps it manageable. In code that calls many fallible functions, you'll write `?` often. That's the cost — and it's a cost paid in characters, not in debugging time.

0.49 Chapter 6: Generics

Our calculator parses and evaluates expressions, and reports errors when the parser hits something it can't handle. But everything is `f64`. What if we wanted integer-only arithmetic? Or complex numbers? Right now we'd have to copy the parser and change every type annotation. That's not engineering — that's a word processor.

Lyric has generics. They look like this:

```
func identity<T>(x: T) -> T {
    return x
}
```

`T` is a type parameter. The compiler generates a specialized copy of `identity` for each concrete type it's called with — `identity<i32>`,

`identity<string>`, `identity<i64>`. No vtables, no boxing, no runtime dispatch. This is monomorphization, the same strategy Rust uses. You pay nothing at runtime.

0.49.1 6.1 Type Parameters

Here's a generic function that returns the larger of two values:

```
func max_val<T: Comparable>(a: T, b: T) -> T {
    if a > b {
        return a
    }
    return b
}
```

The `: Comparable` after `T` is a constraint. It tells the compiler that `T` must support comparison operators. Without it, `a > b` won't typecheck — the compiler doesn't know that `T` has a `>` operator.

Call it with explicit type arguments:

```
let result = max_val<i32>(10, 20)
println(f"max: {result}")  // max: 20
```

Or let the compiler figure it out:

```
let m = max_val(10, 20)
println(f"max(10, 20) = {m}")  // max(10, 20) = 20
```

The compiler sees two `i32` arguments, infers `T = i32`, and generates `max_val_i32`. You only need explicit type arguments when the compiler can't infer them — which in practice is rare.

0.49.2 6.2 Inference

Type inference in Lyric works from arguments to type parameters. The compiler examines each argument's type and unifies it with the corresponding parameter:

```
func identity<T>(x: T) -> T {
    return x
}
```

```
let x = identity(42)            // T = i32
let s = identity("hello")       // T = string
```

This extends to collection types:

```
func first<T>(xs: [T]) -> T? {
    if xs.len() == 0 {
        return null
    }
    return xs[0]
}

let nums: [i32] = [10, 20, 30]
let f = first(nums)
println(f"first([10,20,30]) = {f!}")  // first([10,20,30]) = 10
```

The compiler sees `[i32]` for `xs`, matches it against `[T]`, and infers `T = i32`. The return type becomes `i32?`. Inference also works through lambda return types and multiple type parameters — if a function takes `(xs: [T], f: func(T) -> U) -> [U]`, the compiler infers both `T` and `U` from the arguments.

0.49.3 6.3 Constraints

A bare `<T>` allows any type. That's useful for `identity`, but most generic code needs to do something with `T` — compare it, hash it, print it. Constraints declare what operations a type parameter must support.

The built-in `Comparable` constraint gives you `<`, `>`, `<=`, `>=`. You write it after the type parameter with a colon:

```
func max_val<T: Comparable>(a: T, b: T) -> T {
    if a > b {
        return a
    }
    return b
}
```

If you try `max_val<string>("a", "b")` and `string` doesn't satisfy `Comparable`, the compiler rejects it. The error happens at compile time, not at link time like C++ templates, and not in a wall of angle brackets.

Lyric ships three built-in constraints:

Constraint	Satisfied by	Provides
`Comparable`	numeric types, `string`, `bool`	`< > <= >=`
`Equatable`	numeric types, `string`, `bool`	`== !=`
`Hashable`	`Sym`, numeric types, `bool` (not `string` — see Ch 10)	`get_hash(self) -> u64`

`Hashable` currently declares only `get_hash`. An `equals` method is on the roadmap and is required for hash tables to handle collisions correctly when keys aren't pointer-equal. Today, `Dict` is safest with `Sym` keys, where the intern table guarantees uniqueness — see Chapter 10.

0.49.4 6.4 Where Clauses

For more complex constraints, or when you want the constraint separate from the parameter list, use `where`:

```
func max_val<T>(a: T, b: T) -> T
  where T: Comparable
{
    if a > b {
        return a
    }
    return b
}
```

This is identical in semantics to `<T: Comparable>`. Where clauses become essential when constraints involve multiple type parameters — we'll see that in Chapter 9 with multi-class interfaces like `where DoublyLinked<P, C>`.

0.49.5 6.5 User-Defined Constraints

Any interface can serve as a constraint. Here's a `Printable` interface used to constrain a generic function:

```
pub interface Printable {
    func to_string(self) -> string
}

class Dog {
    name: string

    pub func to_string(self) -> string {
        return self.name
    }
}

func print_it<T: Printable>(item: T) -> string {
    return item.to_string()
}

func main() {
    let d = Dog { name: "Rex" }
    let result = print_it<Dog>(d)
    println(result)  // Rex
}
```

The constraint `T: Printable` means: `T` must be a type that implements a `to_string(self) -> string` method. `Dog` has one, so it satisfies the constraint. This is structural — `Dog` doesn't need to declare `implements Printable` (though it can, as we'll see in Chapter 9). The compiler checks that the required methods exist.

This is Lyric's answer to Rust's trait bounds. But notice the difference: in Rust, you'd write `T: Display + PartialOrd + Clone`. In Lyric, you name the *capability* — `T: Printable`, `T: Comparable`, `T: Hashable`. Each constraint is a meaningful abstraction, not a shopping list of individual operations.

0.49.6 6.6 Type Aliases

When types get long, alias them:

```
type StringList = [string]

func test_aliases() {
```

```
    let names: StringList = ["alice", "bob"]
    println(names.len())         // 2
    println(f"first: {names[0]}")  // first: alice
}
```

`StringList` and `[string]` are interchangeable. The alias is a convenience, not a new type — the compiler treats them identically.

0.49.7 6.7 Union Types

Sometimes a value can be one of several types. Union types are an alternative to generics when you know the exact set of types:

```
func describe(val: string | i32) -> string {
    return match val {
        string => { "it's a string" }
        i32 => { "it's an int" }
    }
}

let a: string | i32 = 42
let b: string | i32 = "hello"
println(describe(a))  // it's an int
println(describe(b))  // it's a string
```

The `match` is exhaustive — the compiler requires a case for each type in the union. If you don't want to handle every type, use a wildcard:

```
func with_default(val: string | i32 | bool) -> string {
    return match val {
        string => { "string" }
        _ => { "other" }
    }
}
```

You can combine them with type aliases for a poor man's `any`:

```
type Any = string | i32 | i64 | f32 | f64 | bool
```

0.49.8 6.8 Monomorphization

We said the compiler generates specialized copies. Here's what that means concretely: `identity<i32>(42)` becomes

`identity_i32(int32_t x)` in the generated C, and `identity<string>("hello` becomes a separate `identity_string(lyric_string x)`. Each call site gets a specialized function with the concrete type baked in. Ten instantiations means ten copies, but the dead code eliminator removes unused ones, and specialized code is often *smaller* because the optimizer can inline with concrete types.

0.49.9 6.9 A Generic Stack

Let's put this together. Here's how a generic stack *would* read — built on slices, with the same shape as the concrete `Stack` from Chapter 3 but parameterized on the element type:

```
class Stack<T> {
    items: [T]

    pub func push(self, item: T) {
        self.items.push(item)
    }

    pub func pop(self) -> T? {
        if self.items.len() == 0 {
            return null
        }
        return self.items.pop()
    }

    pub func peek(self) -> T? {
        if self.items.len() == 0 {
            return null
        }
        return self.items[self.items.len() - 1]
    }

    pub func is_empty(self) -> bool {
        return self.items.len() == 0
    }
}
```

Use it like this:

```
let empty: [f64] = []
let mut stack = Stack<f64> { items: empty }
stack.push(1.0)
stack.push(2.0)
stack.push(3.0)
let top = stack.pop()
println(f"popped: {top!}")  // popped: 3
```

Conceptually, the compiler generates `Stack_f64` with `push_f64`, `pop_f64`, and so on. If we also use `Stack<string>` somewhere, it generates a second complete set. Each is fully specialized — no indirection.

Roadmap: as of this writing, generic class methods that access `self.<field>` lower to a null receiver in the C backend — `s.push(1.0)` on a `Stack<T>` compiles through checking and monomorphization but generates C that segfaults. Generic free functions (like the `max_val<T: Comparable>` and `first<T>` above) work today; generic classes are next on the bring-up list. Also, `Stack<f64> { items: [] }` with an untyped empty literal fails earlier with a type-variable leak, so the example uses a typed `let empty: [f64] = []` to seed inference. Logged in `~/projects/lyric/TODO`.

0.49.10 6.10 Toward a Generic Parser

Our calculator parser from Chapter 5 hardcodes `f64` as the result type. With generics, we could parameterize it — but we'd need a constraint that captures everything a numeric type needs: parsing from strings, a zero value, arithmetic. Rather than a shopping list of individual operations, Lyric lets you define a single named capability:

```
interface NumericParser<T> {
    func T.parse_number(self, s: string) -> (T, error)
    func T.zero(self) -> T
}
```

One constraint instead of Rust's `T: FromStr + Default + Add<Output=T> + Mul<Output=T>`. We'll revisit this in Chapter 9, where multi-class interfaces let us bind an entire parser-evaluator system to any numeric type with a single `impl` block.

For now, our `f64` calculator works. The next chapter adds tests to

make sure it stays working.

0.50 Chapter 7: Testing

```
func test_addition() {
    assert_eq(2 + 2, 4)
}

func test_string_length() {
    assert_eq("hello".len(), 5)
}
```

```
$ lyric test math_test.ly
PASS  test_addition
PASS  test_string_length
2 tests, 2 passed, 0 failed
```

Functions whose names start with `test_` are discovered automatically — the compiler scans the LIR for functions matching the prefix, generates a test runner in the emitted C, and executes them sequentially. No registration, no macros, no `main`. That's the entire testing model.

0.50.1 7.1 assert and assert_eq

Two assertion builtins:

```
assert(condition, message)
assert_eq(actual, expected, message)
```

The message is optional in both. When an assertion fails, it prints file and line automatically:

```
func test_failing() {
    assert_eq(2 + 2, 5, "basic arithmetic")
}
```

```
FAIL  test_failing
  assert_eq failed at math_test.ly:3
    basic arithmetic
    expected: 5
    got:      4
```

`assert_eq` prints both values on failure. Use `assert` for boolean conditions — null checks, bounds checks, invariants:

```
func test_parse_succeeds() {
    let (result, err) = parse("1 + 2")
    assert(err == null, "parse error")
    assert_eq(result, 3.0)
}
```

0.50.2 7.2 Testing the Calculator

Let's test the tokenizer from Chapter 4:

```
func test_tokenize_number() {
    let tokens = tokenize("42")
    assert_eq(tokens.len(), 1)
    assert_eq(tokens[0].kind, TokenKind.Number)
    assert_eq(tokens[0].text, "42")
}

func test_tokenize_operator() {
    let tokens = tokenize("+")
    assert_eq(tokens.len(), 1)
    assert_eq(tokens[0].kind, TokenKind.Plus)
    assert_eq(tokens[0].text, "+")
}

func test_tokenize_expression() {
    let tokens = tokenize("3 + 4 * 2")
    assert_eq(tokens.len(), 5)
    assert_eq(tokens[0].text, "3")
    assert_eq(tokens[1].text, "+")
    assert_eq(tokens[2].text, "4")
    assert_eq(tokens[3].text, "*")
    assert_eq(tokens[4].text, "2")
}
```

No setup, no teardown. Each test creates what it needs. If `tokenize` changes its return type, the tests fail at compile time, not at runtime with a mysterious null pointer.

Now the parser and its error paths:

```
func test_parse_number() {
    let (result, err) = parse("42")
    assert(err == null, "unexpected error")
    assert_eq(result, 42.0)
}

func test_parse_precedence() {
    let (result, err) = parse("3 + 4 * 2")
    assert(err == null, "unexpected error")
    assert_eq(result, 11.0)
}

func test_parse_parentheses() {
    let (result, err) = parse("(3 + 4) * 2")
    assert(err == null, "unexpected error")
    assert_eq(result, 14.0)
}

func test_parse_empty() {
    let (_, err) = parse("")
    assert(err != null, "expected error on empty input")
}

func test_parse_incomplete_expr() {
    let (_, err) = parse("3 + @")
    assert(err != null, "expected error on incomplete expression")
}

func test_error_message() {
    let (_, err) = parse("3 +")
    assert(err != null, "expected error")
    assert_eq(f"{err}", "unexpected end of input")
}
```

Because errors are values, you test them like any other return — check the error, check the value. No exception catching, no panic recovery, no special test syntax.

The `parse()` function used here is the Chapter 5 wrapper that tokenizes and parses in one call — it calls `tokenize`, builds a `Parser`, and returns `(f64, error)`. The last test stringifies the

error with `f"{err}"` rather than calling `err!.message()`: the C backend doesn't yet route `.message()` through interface dispatch on an `error`-typed value, so f-string interpolation is the working idiom (Chapter 5 §5.5). *Roadmap: `error` will get real interface dispatch, after which `err!.message()` will work directly.*

A note on `assert_eq(result, 42.0)`: float equality is exact comparison. This is safe for integer-valued floats, but `parse("1.0 / 3.0 * 3.0")` would fail. For real floating-point tests, compare against an epsilon — write a helper that asserts `|a - b| < tol`. *A built-in `assert_eq_approx(actual, expected, tol)` is on the roadmap; until it lands, the helper is one line.*

0.50.3 7.3 How It Works Under the Hood

When you run `lyric test calculator_test.ly`, the compiler:

1. Parses and compiles the file through the full pipeline — desugar, check, lower, optimize, monomorphize.
2. Scans the LIR for functions with names starting with `test_`.
3. Generates a C `main` that calls each test function in source order, with result tracking.
4. Compiles the C with gcc, runs the binary, reports results.

A failed `assert` or `assert_eq` prints the failure (file, line, message, and for `assert_eq` the expected and got values), terminates *that test*, and the runner moves on to the next one. The suite's exit code is 0 if every test passed, 1 if any failed.

The test runner is generated C. There is no Lyric test framework — the compiler *is* the test framework.

0.50.4 7.4 Testing with Multiple Files

Real programs span multiple files. Pass them all to `lyric test`:

```
$ lyric test test_lexer.ly ../src/lexer.ly ../src/ast.ly
```

You list every source file — the compiler has no build file or import resolution. For module-based projects, `lyric test -mod .` compiles the whole module (see Chapter 13). For small programs, listing files explicitly is simple enough.

The compiler merges all files into a single compilation unit, then discovers `test_` functions from any of them. This is how the Lyric compiler tests itself — `test_lexer.ly` imports the real lexer source:

```
func make_lex(src: string) -> Lexer {
    return new_lexer(src, sym("test.ly"))
}

func test_keywords_statement() {
    let lex = make_lex("let if else for in while match return brea
    assert_eq(next_skip_nl(lex).kind, KLet, "let")
    assert_eq(next_skip_nl(lex).kind, KIf, "if")
    assert_eq(next_skip_nl(lex).kind, KElse, "else")
    assert_eq(next_skip_nl(lex).kind, KFor, "for")
    assert_eq(next_skip_nl(lex).kind, KIn, "in")
    assert_eq(next_skip_nl(lex).kind, KWhile, "while")
    assert_eq(next_skip_nl(lex).kind, KMatch, "match")
    assert_eq(next_skip_nl(lex).kind, KReturn, "return")
    assert_eq(next_skip_nl(lex).kind, KBreak, "break")
    assert_eq(next_skip_nl(lex).kind, KContinue, "continue")
    assert_eq(next_skip_nl(lex).kind, SEOF, "eof")
}
```

That's the real test, quoted verbatim from `testdata/test_lexer.ly`. No mocking, no dependency injection — it creates a real lexer with real source code and checks real tokens. The `sym("test.ly")` call creates a filename symbol for error reporting.

0.50.5 7.5 Auto-Generated to_string

When `assert_eq` fails, it needs to print both values. For built-in types this works automatically. For your own types, the compiler generates `to_string()` for enums, structs, and classes:

```
enum Color { Red Green Blue }

func test_enum_printing() {
    let c = Color.Red
    assert_eq(c, Color.Blue, "color check")
}
```

```
FAIL  test_enum_printing
```

```
  assert_eq failed at color_test.ly:5
    color check
    expected: Blue
    got:      Red
```

You never write display formatting for test output.

0.50.6 7.6 What Lyric Doesn't Have

No mocking framework. No fixtures. No setup/teardown. No coverage reporting. No parameterized tests. No test discovery beyond the `test_` prefix.

This is deliberate. Lyric is designed for compilers and systems tools, where the right testing strategy is: create real inputs, run real code, check real outputs. If your test needs a mock, your code probably needs a better interface.

The Lyric compiler itself has 78 tests across its test files. Every test creates real ASTs, runs real desugar passes, checks real type-checking results. None of them mock anything:

```
func test_field_generates_getter_and_setter() {
    // Triple-quoted strings (""") preserve newlines and embedded
    let src = """lyric t { interface Named<T> { field T.name: stri
    let file = td_parse(src)
    desugar_interface_fields(file)
    let named = file.fb_children()[0].id_children()[0]
    assert(len(named.im_children()) >= 2, "expected getter + sette
    let getter = named.im_children()[0]
    let setter = named.im_children()[1]
    assert(getter.name!.name == "name", "getter name")
    assert(setter.name!.name == "set_name", "setter name")
}
```

This test parses an interface declaration with a `field` injection, runs the desugar pass, and verifies the compiler generated the right methods. It uses triple-quote strings to embed Lyric source inside a Lyric test. The function under test — `desugar_interface_fields` — is the same function the compiler calls during compilation.

0.50.7 7.7 The Calculator Test Suite

Here's the complete test file for our calculator:

```
// calculator_test.ly - tests for the calculator

// Tokenizer tests
func test_tokenize_numbers() {
    let tokens = tokenize("3.14")
    assert_eq(tokens.len(), 1)
    assert_eq(tokens[0].kind, TokenKind.Number)
    assert_eq(tokens[0].text, "3.14")
}

func test_tokenize_parens() {
    let tokens = tokenize("(1 + 2)")
    assert_eq(tokens.len(), 5)
    assert_eq(tokens[0].kind, TokenKind.LeftParen)
    assert_eq(tokens[4].kind, TokenKind.RightParen)
}

// Parser tests
func test_eval_simple() {
    let (result, err) = parse("10 - 3")
    assert(err == null, "no error expected")
    assert_eq(result, 7.0)
}

func test_eval_nested_parens() {
    let (result, err) = parse("((2 + 3) * (4 - 1))")
    assert(err == null, "no error expected")
    assert_eq(result, 15.0)
}

// Error tests
func test_unmatched_paren() {
    let (_, err) = parse("(1 + 2")
    assert(err != null, "expected unmatched-paren error")
}

func test_empty_parens() {
```

```
    let (_, err) = parse("()")
    assert(err != null, "expected empty-parens error")
}
```

```
$ lyric test calculator_test.ly calc.ly
PASS  test_tokenize_numbers
PASS  test_tokenize_parens
PASS  test_eval_simple
PASS  test_eval_nested_parens
PASS  test_unmatched_paren
PASS  test_empty_parens
6 tests, 6 passed, 0 failed
```

Six tests, each one a function, each one checking one thing. The test file is a regular `.ly` file — you can add a `main` function and run it directly if you want. Tests are just functions.

0.51 Chapter 8: Relations — Ownership Without a Borrow Checker

0.51.1 8.1 The Problem

Here's a calculator that builds an AST. Each expression node has children:

```
class Expr {
    kind: string
    value: f64
    left: Expr?
    right: Expr?
}

func make_binop(op: string, left: Expr, right: Expr) -> Expr {
    return Expr { kind: op, value: 0.0, left: left, right: right }
}

func main() {
    let a = Expr { kind: "num", value: 3.0, left: null, right: nul
    let b = Expr { kind: "num", value: 4.0, left: null, right: nul
    let plus = make_binop("+", a, b)
    println(f"built: {plus.kind}")
}
```

This works, but there's a problem hiding in the design. Who owns `a` and `b`? Both `main` and `plus` have references to them. If `plus` is destroyed, should `a` and `b` be destroyed too? If `a` is reassigned, should `plus.left` become dangling? What if we build a tree with thousands of nodes — who cleans them all up?

In C++, you'd write a destructor that walks the tree and deletes children. You'd get it wrong at least once — everyone does. In Rust, you'd use `Box<Expr>` for owned children and fight the borrow checker whenever you need a parent pointer. In Go, you'd let the GC handle it and accept the pauses.

In Lyric, you declare a relation.

0.51.2 8.2 Your First Relation

```
class Team { name: string }
class Player { name: string }

relation ArrayList Team:roster owns [Player:team]
```

That one line — `relation ArrayList Team:roster owns [Player:team]` — tells the compiler everything it needs to know:

- A `Team` owns a dynamic array of `Player` objects.
- The relation type is `ArrayList` — a stdlib interface that provides array-backed storage with O(1) swap-remove.
- The label `roster` prefixes fields injected into `Team`. The label `team` prefixes fields injected into `Player`.
- The keyword `owns` means cascade destroy — when a `Team` is destroyed, all its `Player` children are destroyed too.

The compiler reads this and generates:

- A field `roster_children: [Player]` on `Team`
- Fields `team_parent: Team?` and `team_index: i32` on `Player`
- Methods `Team.roster_append(p)` and `Team.roster_remove(p)` (label-prefixed from the parent label `roster`)
- A destructor on `Team` that cascade-destroys all children
- A destructor on `Player` that removes itself from its parent's array

- An impl block that wires the `ArrayList` interface fields to these concrete fields

You don't write any of that. Here's the full program:

```
class Team { name: string }
class Player { name: string }

relation ArrayList Team:roster owns [Player:team]

func main() {
    let t = Team { name: "Wolves" }
    let p1 = Player { name: "Alice" }
    let p2 = Player { name: "Bob" }
    let p3 = Player { name: "Carol" }

    t.roster_append(p1)
    t.roster_append(p2)
    t.roster_append(p3)

    println(t.roster_children.len())
    println(p1.team_index)
    println(p2.team_index)
    println(p3.team_index)

    // Remove middle element (Bob) - Carol should swap into Bob's
    t.roster_remove(p2)
    println(t.roster_children.len())
    println(p3.team_index)

    // Parent destroy - cascade
    let t2 = Team { name: "Bears" }
    let p4 = Player { name: "Dan" }
    t2.roster_append(p4)
    t2.destroy()
    println(isnull(p4.team_parent))
}
```

Output:

```
3
0
```

```
1
2
2
1
true
```

Three players appended — indices 0, 1, 2. Remove Bob (index 1) — Carol swaps down from index 2 to index 1, array shrinks to length 2. Destroy `t2` — Dan's parent becomes null because `owns` means cascade destroy. Accessing `p4` after this is technically a use-after-free; in practice, the slab allocator zeros freed memory so `isnull(p4.team_parent)` returns `true`. Don't rely on this — it's undefined behavior. We'll discuss this further in Chapter 11.

`t.roster_append(p1)` and `t.roster_remove(p2)` are the methods the `relation` declaration generated. The label `roster` on the parent side of the relation prefixes the method names — that's why it's `roster_append`, not just `append`. If you'd declared `relation ArrayList Team:squad owns [Player:team]` instead, the methods would be `t.squad_append(...)` and `t.squad_remove(...)`. The label keeps method names from colliding when one class participates in several relations (see §8.5).

The methods aren't generated from thin air. They come from a stdlib interface called `ArrayListBase`, which is what the `ArrayList` hint embeds. Here's the relevant part of `stdlib/std.ly`:

```
pub interface ArrayListBase<P, C> {
    field P.children: [C]
    field C.parent: P?
    field C.index: i32

    // Method-style append: p.append(c)
    pub trusted func P.append(self, child: C) {
        ref child
        child.index = len(self.children)
        child.parent = self
        let mut kids = self.children
        kids.push(child)
        self.children = kids
    }
```

```
    // Method-style remove: p.remove(c)
    pub trusted func P.remove(self, child: C) {
        let idx = child.index
        let kids = self.children
        let last_idx: i32 = len(kids) - 1
        if idx < last_idx {
            let last_child = kids[last_idx]
            last_child.index = idx
            kids[idx] = last_child
        }
        self.children = kids[0:last_idx]
        child.parent = null
        child.index = 0
        unref child
    }
}

pub interface ArrayList<P, C> {
    embed ArrayListBase<P, C>

    destructor P {
        let kids = self.children
        let mut i: i32 = len(kids) - 1
        while i >= 0 {
            kids[i].parent = null
            kids[i].destroy()
            i = i - 1
        }
    }

    destructor C {
        array_remove<P, C>(self)
    }
}
```

`ArrayListBase` declares the fields and the append/remove operations. `ArrayList embeds` it — copying the fields in — and adds the destructors that make it an *owning* relation. There's a sibling interface `RefArrayList` that embeds the same base but uses non-cascading destructors; we'll see the linked-list analogue (`OwningList`

vs `RefList`) in §8.3.

`pub func P.append(self, child: C)` is a method bound to whatever class plays `P`. When `relation ArrayList Team:roster owns [Player:team]` binds `Team` as `P` and `Player` as `C`, the desugar pass copies this method onto `Team` with the parent label as prefix — so `Team.append` becomes `Team.roster_append`. That's where `t.roster_append(p1)` comes from. The `roster_remove` method is the same story. The `array_remove<P, C>` call inside `destructor C` is the *free-function* form of the same operation, lifted from `ArrayListBase` unchanged — both forms exist, and Lyric's UFCS rule means `t.roster_append(p)` and `array_append<Team, Player>(t, p)` lower to the same generated code. The book prefers the method form.

The `array_remove` body uses swap-remove for O(1) deletion — the last element swaps into the removed slot, then the slice shrinks by one. **Don't cache array indices across removals** — swap-remove changes the index of whatever element used to be at the end.

The `trusted` keyword on the methods opens a small window where the interface can call the raw `ref child` / `unref child` ops to manage the child's reference count by hand — these are the back-pointer fix-ups that make the relation's lifetime contract work. You don't write `trusted` code yourself unless you're building your own container interface; the four stdlib relation types are the only ones most programs need.

The interface is generic over `P` and `C`. It works for any parent-child pair. The `relation` declaration tells the compiler which concrete types to bind — `Team` as `P`, `Player` as `C` — and auto-generates the field bindings so that when `ArrayListBase` code accesses `self.children`, it reaches `Team.roster_children`.

0.51.3 8.3 owns vs refs

`owns` means cascade destroy — when the parent dies, its children die with it. But sometimes you want references without ownership. A `Room` might track its current `Guest` objects, but destroying a room shouldn't destroy the guests:

```
class Room { name: string }
class Guest { name: string }

relation RefList Room:room refs [Guest:guest]

func main() {
    let r = Room { name: "Lobby" }
    let g1 = Guest { name: "Alice" }
    let g2 = Guest { name: "Bob" }
    let g3 = Guest { name: "Carol" }

    r.room_append(g1)
    r.room_append(g2)
    r.room_append(g3)

    // Walk the list
    let mut cur = r.room_first
    while !isnull(cur) {
        println(cur!.name)
        cur = cur!.guest_next
    }

    // Remove middle element - Bob is unlinked but still alive
    r.room_remove(g2)
    println("after remove:")
    cur = r.room_first
    while !isnull(cur) {
        println(cur!.name)
        cur = cur!.guest_next
    }

    // Destroy parent - children should be unlinked but still aliv
    r.destroy()
    println(f"g1 parent null: {isnull(g1.guest_parent)}")
    println(f"g3 parent null: {isnull(g3.guest_parent)}")
    // Children still accessible - refs means no cascade
    println(f"g1 name: {g1.name}")
    println(f"g2 name: {g2.name}")
    println(f"g3 name: {g3.name}")
}
```

Output:

```
Alice
Bob
Carol
after remove:
Alice
Carol
g1 parent null: true
g3 parent null: true
g1 name: Alice
g2 name: Bob
g3 name: Carol
```

`refs` instead of `owns`. When the room is destroyed, all three guests are unlinked — their parent pointers become null — but they survive. Bob is doubly interesting: we explicitly removed him from the room with `r.room_remove(g2)` before the destroy, and he's still alive at the end because nobody owns him.

The methods this time are `r.room_append(...)` and `r.room_remove(...)` — same label-prefix rule as `ArrayList`, just bound to a doubly-linked-list interface. Walking the list uses the injected fields directly: `r.room_first` (the list head) and `cur!.guest_next` (each child's forward pointer). The `room` and `guest` labels prefix the fields the same way they prefix the methods.

The `RefList` destructor walks the linked list and nulls out all the pointers, but doesn't call `.destroy()` on the children. Compare with the `OwningList` destructor, which does call `.destroy()`:

```
// From stdlib - OwningList destructor
destructor P {
    let mut cur = self.first()
    while !isnull(cur) {
        let next = cur!.next()
        cur!.set_parent(null)
        cur!.destroy()    // cascade destroy
        cur = next
    }
}
```

```
// From stdlib - RefList destructor
destructor P {
    let mut cur = self.first()
    while !isnull(cur) {
        let next = cur!.next()
        cur!.set_parent(null)
        cur!.set_prev(null)
        cur!.set_next(null)  // unlink only
        cur = next
    }
    self.set_first(null)
    self.set_last(null)
}
```

Both `OwningList` and `RefList` embed `DoublyLinked<P, C>`, which provides the linked-list fields (`first`, `last`, `next`, `prev`, `parent`) and both forms of append/remove — the free-function form (`dll_append`, `dll_remove`) and the method form (`P.append`, `P.remove`) we used above. The difference between `OwningList` and `RefList` is purely in the destructors.

0.51.4 8.4 The Four Relation Types

Lyric's standard library provides four relation types:

Type	Storage	Destruction	Use case
`ArrayList`	Dynamic array	Cascade (owns)	Most parent-child relationships
`OwningList`	Doubly-linked list	Cascade (owns)	When insertion order matters, frequent middle removal
`RefList`	Doubly-linked list	Unlink (refs)	References without ownership
`HashedList`	Hash table	Cascade (owns)	Keyed lookup by hash

`ArrayList` is the default choice. Dynamic array, O(1) swap-remove,

compact memory. Use `OwningList` when you need stable iteration order during removal — a linked list won't shuffle elements around. Use `RefList` for non-owning references. Use `HashedList` when you need lookup by key, which is how `Dict` is built (Chapter 10).

All four are written in Lyric, defined in the standard library. None of them are compiler builtins. The `relation` keyword and the `field`/`destructor`/`embed` machinery are the builtins — the data structures are just interfaces that use them.

And there's nothing magic about *these* four interfaces. The `relation` declaration accepts **any binary interface** (one with two type parameters in `(parent, child)` order) as its hint — including ones you write yourself. We'll see how to build one in Chapter 9.

0.51.5 8.5 Multiple Relations

A class can participate in multiple relations. Here's a parent with two kinds of children:

```
class Child1 { val: i32 }
class Child2 { val: i32 }
class Parent { name: string }

relation ArrayList Parent:c1 owns [Child1:c1]
relation ArrayList Parent:c2 owns [Child2:c2]

func main() {
    let p = Parent { name: "test" }
    let a = Child1 { val: 1 }
    let b = Child2 { val: 2 }
    p.c1_append(a)
    p.c2_append(b)
    print(p.c1_children.len())
    print(p.c2_children.len())
    p.c1_remove(a)
    print(p.c1_children.len())
    print(p.c2_children.len())
}
```

Output: `1101`

The labels (`c1` and `c2`) keep both the method names and the field names from colliding: `p.c1_append` vs `p.c2_append`, `c1_children` vs `c2_children`, `c1_parent` vs `c2_parent`. Each relation is independent — removing a `Child1` doesn't affect the `Child2` collection. The output reads left to right (we used `print`, not `println`): after both appends, c1 has 1 child, c2 has 1 child. After removing the `Child1`, c1 has 0, c2 still has 1.

A child can also belong to multiple parents. In `destroy_shared.ly`, a `Player` belongs to both `TeamA` and `TeamB` via separate `OwningList` relations. Destroying `TeamA` cascade-destroys the player, which automatically removes itself from `TeamB`:

```
class TeamA { name: string }
class TeamB { name: string }
class Player { name: string }

relation OwningList TeamA:team_a owns [Player:pa]
relation OwningList TeamB:team_b owns [Player:pb]

func main() {
    let a = TeamA { name: "Alphas" }
    let b = TeamB { name: "Betas" }
    let p = Player { name: "Alice" }

    a.team_a_append(p)
    b.team_b_append(p)

    println(f"a has player: {!isnull(a.team_a_first)}")
    println(f"b has player: {!isnull(b.team_b_first)}")

    // Destroy team A - cascade-destroys Alice,
    // which auto-removes her from team B
    a.destroy()
    println(f"b has player after destroy: {!isnull(b.team_b_first)
}
```

Output:

```
a has player: true
b has player: true
b has player after destroy: false
```

Alice was in both teams. Destroying TeamA triggers Alice's destructor, which (from `OwningList`'s `destructor C`) unlinks her from every list she's in — including TeamB's. TeamB's list is now empty — no dangling pointers, no manual cleanup.

0.51.6 8.6 An AST with Relations

Let's bring this back to the calculator. Instead of manual `Expr` nodes with nullable children, we can use relations to express the tree structure:

```
class Program { name: string }
class Stmt { kind: string }
class Expr {
    kind: string
    value: f64
    op: string
}

relation ArrayList Program:stmts owns [Stmt:prog]
relation ArrayList Stmt:args owns [Expr:stmt]
relation ArrayList Expr:operands owns [Expr:parent_expr]

func main() {
    let prog = Program { name: "calc" }

    // Build: 3 + 4
    let add = Expr { kind: "binop", value: 0.0, op: "+" }
    let three = Expr { kind: "num", value: 3.0, op: "" }
    let four = Expr { kind: "num", value: 4.0, op: "" }

    add.operands_append(three)
    add.operands_append(four)

    let print_stmt = Stmt { kind: "print" }
    print_stmt.args_append(add)
    prog.stmts_append(print_stmt)

    // Walk the tree
    let stmt = prog.stmts_children[0]
```

```
    let expr = stmt.args_children[0]
    println(f"stmt: {stmt.kind}")
    println(f"expr: {expr.kind} {expr.op}")
    println(f"left: {expr.operands_children[0].value}")
    println(f"right: {expr.operands_children[1].value}")

    // Destroy the whole tree in one call
    prog.destroy()
}
```

Notice that `Expr` is both a parent (of `operands`) and a child (of `Stmt` via `args`). A class can play either role in any number of relations; each relation's labels keep the injected fields and methods separate. `add.operands_append(three)` reaches the `operands` relation; `print_stmt.args_append(add)` reaches the `args` relation; both work on the same `Expr` instance without ambiguity.

`prog.destroy()` destroys the program, which cascade-destroys all statements, which cascade-destroy all expressions, which cascade-destroy their operands. The entire tree is cleaned up deterministically, in reverse order, with no manual traversal and no GC.

This is what the Lyric compiler itself does. The AST — 33,500 lines of Lyric source — uses relations throughout. `File` owns `Block`, `Block` owns `FuncDecl`, `FuncDecl` owns `Stmt`, and so on. One call to `file.destroy()` cleans up the entire compilation unit.

0.51.7 8.7 `final` Functions: User Code at Destruction Time

Relations handle memory. But some destruction work isn't about memory — it's about external resources. A class that holds a file handle, a network connection, or a lock needs to release that resource *before* the auto-generated destructor runs. Lyric calls these `final` functions:

```
class Connection {
    name: string
    final func cleanup(self) {
        println(f"closing {self.name}")
    }
}
```

```
func main() {
    let c = Connection { name: "db" }
    println("doing work")
    // c goes out of scope; cleanup runs, then slab is freed.
}
```

Output:

```
doing work
closing db
```

The `final` keyword marks `cleanup` as the class's pre-destruction hook. When `c` is about to be destroyed, Lyric calls `cleanup(self)` first, then runs the auto-generated destructor (which handles relations and frees the slab slot). The execution order is fixed:

```
final func  →  relation destructors (cascade + unlink)  →  slab fr
```

A class can have only one `final` function. Use it for the things the compiler can't infer — closing OS handles, flushing buffers, removing yourself from some external registry. Use relations for the parent-child memory ownership the compiler *can* infer.

If you call `c.destroy()` explicitly, today's compiler fires the `final` function twice — once at the explicit call, once again at scope exit. The workaround is to let scope exit drive destruction (omit the explicit `c.destroy()`) until the compiler grows a one-shot guard.

You won't write `final` functions often. Most classes only own other Lyric objects, and relations handle that for free. `final` is the escape hatch for when destruction has to reach out of the language — into the operating system, the network, or another process.

0.51.8 8.8 The Trade-Off

Relations don't prevent use-after-free at compile time — if you hold a reference to a destroyed object, you'll crash. The trade-off is deliberate. We proved over 30 years of EDA tools processing billions of objects that this almost never happens when the ownership graph is explicit. The bugs come from *implicit* ownership — when you can't see who owns what. Relations make it visible.

The next chapter shows how the interfaces behind relations work — `field` injection, `embed`, `destructor`, and the impl blocks that wire

everything together.

0.52 Chapter 9: Interfaces — Multi-Class Contracts

Chapter 8 showed what relations *do*. This chapter shows how they work. The `ArrayList`, `OwningList`, `RefList`, and `HashedList` from the standard library aren't compiler builtins — they're interfaces written in Lyric, using the same features available to you.

0.52.1 9.1 The Multi-Class Problem

```
interface Graph<G, N, E> {
    func G.nodes(self) -> [N]
    func N.out_edges(self) -> [E]
    func E.tgt_node(self) -> N

    pub func count_edges(graph: G) -> i32 {
        let mut total: i32 = 0
        let nodes = graph.nodes()
        let mut i: i32 = 0
        while i < nodes.len() {
            let edges = nodes[i].out_edges()
            total = total + edges.len() as i32
            i = i + 1
        }
        return total
    }
}
```

Three type parameters. Methods bound to each one: `G` has `.nodes()`, `N` has `.out_edges()`, `E` has `.tgt_node()`. And `count_edges` is a *default method* — a generic algorithm written once, specialized per binding.

Unlike Go and Rust, Lyric interfaces constrain multiple types simultaneously. Go interfaces constrain one type. Rust traits use associated types to link them, but can't express a single constraint spanning three independent types. Haskell's multi-parameter typeclasses are the closest analogue, but Lyric does this with zero runtime cost via monomorphization. This is from `testdata/interfaces.ly` — it compiles and runs.

0.52.2 9.2 Impl Blocks: Wiring Concrete Types

The `Graph` interface doesn't know about any concrete types. To use it, you bind concrete classes via an `impl` block:

```
class SimpleGraph {
    node_list: [SimpleNode]

    pub func get_nodes(self) -> [SimpleNode] {
        return self.node_list
    }
}

class SimpleNode {
    name: string
    edges: [SimpleEdge]

    pub func get_edges(self) -> [SimpleEdge] {
        return self.edges
    }
}

class SimpleEdge {
    target: SimpleNode

    pub func get_target(self) -> SimpleNode {
        return self.target
    }
}

impl Graph<SimpleGraph, SimpleNode, SimpleEdge> {
    G.nodes = SimpleGraph.get_nodes
    N.out_edges = SimpleNode.get_edges
    E.tgt_node = SimpleEdge.get_target
}
```

The impl block says: `SimpleGraph` plays the role of `G`, `SimpleNode` plays `N`, `SimpleEdge` plays `E`. The method aliases map interface methods to concrete methods — `G.nodes` becomes `SimpleGraph.get_nodes`.

Now `count_edges` works:

```
let n2 = SimpleNode { name: "B", edges: [] }
let e1 = SimpleEdge { target: n2 }
let n1 = SimpleNode { name: "A", edges: [e1] }
let g = SimpleGraph { node_list: [n1, n2] }
let count = count_edges<SimpleGraph, SimpleNode, SimpleEdge>(g)
println(count)  // 1
```

The three type parameters are explicit because the compiler can't always infer them — a class could participate in multiple `Graph` implementations. Monomorphization generates a `count_edges` specialized for these three concrete types. No vtables, no dynamic dispatch. The generated C code is a direct function call.

Roadmap: the spec promises method-call syntax on default methods too — `g.count_edges()` once the interface is wired up on `G`. Today the checker only resolves the free-function form `count_edges<G, N, E>(g)`, so that's what the examples use.

Classes can also declare which interfaces they satisfy with `implements`:

```
class Task implements Displayable, Prioritizable {
    name: string
    priority: i32
}
```

This is documentation and a compiler check. Lyric uses structural interface satisfaction by default — if the methods exist, the interface is satisfied. `implements` just makes it explicit.

Today, `implements` is declarative only — the checker records the claim but doesn't yet verify that the required methods are actually present. Missing methods surface as errors later in lowering or codegen instead of at the declaration site. The roadmap item is to do the structural check up front so you get a clean "Task: method `display` required by Displayable is missing" error.

0.52.3 9.3 Default Methods and Field Accessors

The `Graph` example uses method aliases — the concrete class already has a method, and the impl block maps the interface name to it. The other style is to give the interface a body — a *default method* — that calls the abstract methods and field accessors it declares:

```
pub interface MyList<P, C> {
    field P.items: [C]
    field C.owner: P?
    field C.pos: i32

    pub func add(parent: P, child: C) {
        let kids = parent.items()
        let num: i32 = kids.len() as i32
        child.set_pos(num)
        child.set_owner(parent)
        parent.set_items(append(kids, child))
    }

    pub func count(parent: P) -> i32 {
        return parent.items().len() as i32
    }
}
```

Two things are happening here. Each `field` declaration auto-generates a getter (`parent.items()`) and a setter (`parent.set_items(...)`) on the type parameter — that's what the default method calls, because inside the interface body the compiler doesn't know `P` will be `Panel` yet. And `add` and `count` are *default methods*: top-level generic functions with a `where MyList<P, C>` clause, written once and specialized per binding.

Bind it to concrete classes with a relation:

```
class Widget { label: string }
class Panel {}

relation MyList Panel:w owns [Widget:p]

func main() {
    let panel = Panel {}
    let w1 = Widget { label: "button" }
    let w2 = Widget { label: "text" }

    add<Panel, Widget>(panel, w1)
    add<Panel, Widget>(panel, w2)
    println(count<Panel, Widget>(panel))  // 2
```

```
}
```

The call site uses the free-function form `add<Panel, Widget>(panel, w1)`. The compiler monomorphizes `add` against `(Panel, Widget)`, rewrites `parent.items()` to read the relation-injected field `panel.w_items`, and emits a direct call — no vtables, no dispatch.

Roadmap: the spec says default methods should also be callable in method-syntax form prefixed by the relation's parent label — `panel.w_add(w1)` and `panel.w_count()` for the relation above. Today only the free-function form resolves for user-defined hints; the relation-method machinery you saw in Chapter 8 (`team.roster_append(p)`, `dir.files_append(f)`) is wired up for the stdlib hints `ArrayList`, `OwningList`, `RefList`, `HashedList`. So if you want method-call ergonomics today, lean on the stdlib hints; default-method method-call on your own interfaces is on the way.

0.52.4 9.4 Field Injection

Chapter 8 showed the *effect* of field injection — `relation ArrayList Team:roster owns [Player:team]` adds `roster_children` to `Team` and `team_parent` to `Player`. Now you can see the mechanism: the `ArrayList` interface declares `field P.children: [C]`, `field C.parent: P?`, and `field C.index: i32`. The desugar pass physically adds these fields to the concrete classes, prefixed with the label from the relation declaration.

The impl block can also bind injected fields to existing fields using `<->`:

```
impl DoublyLinked<Folder, File> {
    P.children <-> Folder.items
    C.label <-> File.title
}
```

This tells the compiler: when the `DoublyLinked` interface accesses `P.children`, use `Folder.items` instead of injecting a new field. You'd use this when `Folder` already has an `items` field — perhaps from an earlier version of your code, or because the field name carries domain meaning that `children` doesn't.

0.52.5 9.5 Destructors

Interfaces can inject destructors into implementing classes:

```
pub interface ArrayList<P, C> {
    // ...
    destructor P {
        let mut i = self.children.len() - 1
        while i >= 0 {
            self.children[i].destroy()
            i = i - 1
        }
    }

    destructor C {
        array_remove(self)
    }
}
```

`destructor P` is injected into whatever concrete class plays `P`. When you call `team.destroy()`, this code runs — iterating the children array in reverse so that children added last are cleaned up first (matching C++ RAII conventions). `destructor C` calls `array_remove` to unlink the child before it's freed.

Destructors cascade. When `team.destroy()` runs, it calls `player.destroy()` for each player. If `Player` is itself a parent in another relation, that destructor fires too. The compiler chains them automatically, in the order the relations were declared.

0.52.6 9.6 Embed

`OwningList` and `RefList` both need linked-list fields and traversal operations. They differ only in their destructors — `OwningList` cascade-destroys children, `RefList` just unlinks them. The common behavior lives in `DoublyLinked`:

```
pub interface DoublyLinked<P, C> {
    field P.first: C?
    field P.last: C?
    field C.next: C?
    field C.prev: C?
```

```
    field C.parent: P?

    pub func dll_append(parent: P, child: C) {
        // ... linked list insertion
    }

    pub func dll_remove(child: C) {
        // ... linked list removal
    }
}
```

`OwningList` embeds it:

```
pub interface OwningList<P, C> {
    embed DoublyLinked<P, C>

    destructor P {
        let mut cur = self.first()
        while !isnull(cur) {
            let next = cur!.next()
            cur!.set_parent(null)
            cur!.destroy()    // cascade destroy
            cur = next
        }
    }

    destructor C {
        // ... unlink from list
    }
}
```

`embed` copies **fields and destructors** from `DoublyLinked` into `OwningList`. Methods stay abstract bindings — their concrete behavior comes from the impl block at instantiation time, or from a separate `where DoublyLinked<P, C>` constraint on a generic function. After expansion, `OwningList` has `first`, `last`, `next`, `prev`, `parent` fields as if they had been declared directly. The desugar pass expands embeds first, before processing anything else. This is why Chapter 8's `OwningList` relations get `first`, `last`, `next`, `prev`, and `parent` fields even though `OwningList` doesn't declare them directly. *The current desugar over-copies — it also drags methods*

across the `embed`, which is how stdlib's `dll_append/dll_remove` show up on `OwningList` relations today. The intended semantics are fields-and-destructors only; expect the over-copy to be removed and the stdlib factoring to grow explicit `where DoublyLinked<P, C>` constraints in its place.

0.52.7 9.7 Where Clauses on Functions

You can write generic functions constrained by an interface using `where`:

```
pub func count_children<P, C>(p: P) -> i32 where ArrayList<P, C> {
    let kids = p.children()
    return kids.len() as i32
}
```

This function works with *any* parent/child pair that implements `ArrayList`. The where clause gives the function access to all of `ArrayList`'s methods and fields. At the call site, you supply concrete types:

```
let team = Team { name: "Warriors" }
let num = count_children<Team, Player>(team)
```

Monomorphization generates a version specialized for `Team` and `Player`. The `where` clause is checked at compile time — if `Team`/`Player` don't have an `ArrayList` impl block, the checker rejects it.

0.52.8 9.8 External Methods

Methods in Lyric don't have to live inside the class body. You can define them externally:

```
func Sym.equals(self, other: Sym) -> bool {
    return self.hash == other.hash
}
```

`func Sym.equals(self, ...)` — an external method on `Sym`. Called with normal method syntax: `s1.equals(s2)`. This is how the standard library adds interface methods to classes without modifying the class declaration. `Dict.set`, `Dict.get`, `Dict.has` — all external methods:

```
pub func Dict.set<K, V>(self, key: K, value: V) where K: Hashable
    // ...
}

pub func Dict.get<K, V>(self, key: K) -> DictEntry<K, V>? where K:
    // ...
}
```

External methods with where clauses and generics — the full power of the type system, applied outside the class definition. This is what makes interfaces composable. A class doesn't need to know about every interface it will satisfy. The interface and the impl block can be defined elsewhere.

0.52.9 9.9 How the Compiler Processes Interfaces

The desugar pipeline runs five passes in a fixed order:

1. **Embeds** — expand `embed` declarations, copying fields and destructors (methods stay abstract bindings — the current desugar over-copies methods too; see §9.6)
2. **Interface fields** — inject `field` declarations into concrete classes
3. **Relations** — process `relation` declarations, binding interfaces to class pairs
4. **Destructors** — inject `destructor` blocks into classes
5. **Default impls** — copy default method bodies, substituting concrete types

Order matters. Embeds must run before interface fields, because embedded fields need to exist before they can be injected. Relations must run before destructors, because relation declarations determine which destructors to inject. Default impls run last, because they need all fields and destructors already in place.

After desugar, the checker sees only concrete classes with concrete fields and methods. It has no idea interfaces were involved. This is the key insight: interfaces are a *compile-time* mechanism. They generate code, then disappear. The runtime never sees an interface, never does dynamic dispatch, never pays for abstraction.

0.52.10 9.10 The Standard Library Is the Proof

Every collection type in Lyric's standard library is built with interfaces and relations:

- `ArrayList<P, C>` — field injection + destructors + `array_append`/`array_remove`
- `DoublyLinked<P, C>` — field injection + `dll_append`/`dll_remove`
- `OwningList<P, C>` — embeds `DoublyLinked`, adds cascade destructors
- `RefList<P, C>` — embeds `DoublyLinked`, adds unlink-only destructors
- `HashedList<P, C>` — field injection + hash table operations + destructors
- `Dict<K, V>` — uses `HashedList` internally (Chapter 10)
- `Hashable` — single-method constraint for hash table keys

733 lines of Lyric in `std.ly` alone (991 including `string.ly`). No compiler magic, no special-cased types. If you don't like how `ArrayList` works, you can write your own — using the same `interface`, `field`, `destructor`, and `embed` that the standard library uses.

This is what it means when we say the standard library *is* the language. The language provides the mechanism. The library provides the policy. You can change the policy.

0.53 Chapter 10: Sym and Dict — Hash Tables Done Right

Every nontrivial program needs a hash table. The calculator's variable bindings, a compiler's symbol table, a configuration file's key-value pairs — all map names to values. Most languages give you a built-in map type. Lyric gives you `Dict`, which is not built in. It's written in Lyric, using the same relations and interfaces from Chapters 8 and 9.

But before we get to `Dict`, we need to talk about the key.

0.53.1 10.1 Sym — Interned Symbols

```
func main() {
    let s1 = sym("hello")
```

```
    let s2 = sym("world")
    let s3 = sym("hello")

    println(s1.get_name())
    println(s2.get_name())

    // Same string should produce same hash
    if s1.get_hash() == s3.get_hash() {
        println("hashes match")
    }

    // Different strings should produce different hashes
    if s1.get_hash() != s2.get_hash() {
        println("hashes differ")
    }
}
```

Output:

```
hello
world
hashes match
hashes differ
```

`sym("hello")` returns a `Sym` — an interned symbol. The hash is computed once at creation and stored as a `u64`. Every subsequent lookup uses that integer — no re-hashing, no touching the string bytes again. In a compiler that looks up identifiers hundreds of thousands of times, this is the difference between hashing the same bytes in a loop and comparing a single integer. Call `sym("hello")` again and you get the same instance — not a copy with the same hash, but the same object. Sym equality is reference equality.

The implementation is in the standard library. Here's the actual code:

```
pub class Sym {
    name: string
    hash: u64

    pub func get_name(self) -> string { return self.name }
    pub func get_hash(self) -> u64 { return self.hash }
```

```
    pub func hash_key(self) -> u64 { return self.hash }
}

pub permanent class SymTable { }
relation HashedList SymTable:st owns [Sym:st]

let mut _sym_table: SymTable? = null

func _get_sym_table() -> SymTable {
    if isnull(_sym_table) {
        _sym_table = SymTable { }
    }
    return _sym_table!
}

pub func sym(name: string) -> Sym {
    let h = hash_string(name)
    let table = _get_sym_table()
    let existing = hash_lookup<SymTable, Sym>(table, h)
    if !isnull(existing) {
        return existing!
    }
    let s = Sym { name: name, hash: h }
    hash_insert<SymTable, Sym>(table, s)
    return s
}
```

The global `SymTable` is itself a `HashedList` relation — the same hash table interface from Chapter 8. `hash_string` (a stdlib builtin using FNV-1a) computes the hash, `hash_lookup` checks if we've seen this string before, and if not, `hash_insert` adds a new `Sym` to the table. The `permanent` keyword on `SymTable` tells the slab allocator never to free instances of this class — interned symbols outlive every function that uses them, by design.

About `HashedList` collisions: today `HashedList` matches entries by `hash_key()` value alone — `Hashable` declares only `get_hash`, with no `equals`. For `Sym`, that's safe because the intern table guarantees one entry per unique string. For other key types, two values that happen to hash to the same `u64` would collide silently. The roadmap fix is to restore `equals` to `Hashable` so the table can disambiguate;

until then, prefer `Sym` keys (which is why the language pushes you toward `sym()` and the backtick form).

Lyric also has backtick syntax for common symbols:

```
let a = `hello`   // same as sym("hello")
let b = `hello`
if a == b {
    println("sym interning works")
}
println(a.get_name())
```

Output:

```
sym interning works
hello
```

The backtick is syntactic sugar. `` `hello` `` compiles to `sym("hello")`. The Lyric compiler uses it throughout for keyword and operator symbols — `` `if` ``, `` `let` ``, `` `+` `` — because it's terse and visually distinct from string literals.

0.53.2 10.2 The Hashable Interface

`Dict` needs its keys to be hashable. The `Hashable` interface is one method:

```
pub interface Hashable {
    func get_hash(self) -> u64
}
```

`Sym` satisfies this — it has `get_hash`. But `string` deliberately does *not*. This is a design decision, not an oversight. If strings were hashable, you could use them as dict keys directly, and you'd be back to re-hashing on every lookup. By requiring `Sym`, we force the hash-once discipline.

If you're building a hash table keyed by something other than strings — say, integer IDs — you implement `Hashable` on your key type:

```
class NodeId {
    id: i32

    pub func get_hash(self) -> u64 {
```

```
        return self.id as u64
    }
}
```

Now `NodeId` can be a `Dict` key.

0.53.3 10.3 Dict — The Hash Table

```
func main() {
    let d = Dict<Sym, i32>()

    d.set(`x`, 10)
    d.set(`y`, 20)
    d.set(`z`, 30)

    // Lookup
    let ex = d.get(`x`)
    if !isnull(ex) {
        println(f"x = {ex!.value}")
    }

    // Has
    if d.has(`y`) {
        println("has y")
    }
    if !d.has(`w`) {
        println("no w")
    }

    // Keys
    let keys = d.keys()
    println(f"key count = {keys.len()}")

    // Remove
    d.remove(`y`)
    if !d.has(`y`) {
        println("y removed")
    }
}
```

Output:

```
x = 10
has y
no w
key count = 3
y removed
```

`Dict<Sym, i32>()` creates an empty hash table mapping `Sym` keys to `i32` values. The API: `.set(key, value)` inserts or replaces, `.get(key)` returns `DictEntry<K, V>?`, `.has(key)` checks existence, `.remove(key)` deletes, `.keys()` returns all keys.

Notice that `.get()` returns a `DictEntry`, not the value directly. You access the value through `.value`:

```
let entry = d.get(`x`)
if !isnull(entry) {
    let val = entry!.value    // the i32
    let key = entry!.key      // the Sym
}
```

This is because `Dict` is built on `HashedList`, which stores children — and a `DictEntry` is that child. There's no wrapper to extract just the value. It's one extra field access, and it gives you the key for free when you need it.

0.53.4 Dict Literals

For dictionaries you know up front, there's a brace-literal shorthand. The parser disambiguates a Dict literal from a struct literal by looking at the first key form:

```
let names = {`alice`: 1, `bob`: 2}      // Dict<Sym, i32>
let nums  = {1: "one", 2: "two"}        // Dict<i32, string>
```

An empty dictionary literal needs a type annotation so the compiler knows what `K` and `V` are:

```
let empty: Dict<Sym, string> = {}
```

The auto-import pass adds the `Dict` class to the compilation unit whenever it sees a Dict literal — you don't write an `import` for it.

Roadmap: the spec lists string-literal keys as a third form (`{"NYC": 8_000_000, "SF": 875_000}`), but the parser doesn't recognize that

shape today — it commits to a struct-literal interpretation when the opening token is a string and trips at the closing brace. Until that's fixed, build string-keyed dictionaries with `let d = Dict<string, i32>()` *and explicit* `d.set(...)` *calls.*

About collisions: today `HashedList` *matches entries by* `hash_key()` *value alone —* `Hashable` *declares only* `get_hash`*. For* `Sym` *keys this is safe (the intern table guarantees one entry per unique string); for other key types, two values that happen to hash the same would collide silently. The roadmap fix is to restore an* `equals` *method to* `Hashable`*. Until then, prefer* `Sym` *keys.*

0.53.5 10.4 How Dict Works

`Dict` is not a compiler builtin. It's two classes and a relation:

```
pub class DictEntry<K, V> where K: Hashable {
    key: K
    value: V

    pub func hash_key(self) -> u64 {
        return self.key.get_hash()
    }
}

pub class Dict<K, V> where K: Hashable { }
relation HashedList Dict<K, V>:d owns [DictEntry<K, V>:d]
```

That's it. `Dict` is an empty class that owns `DictEntry` children via `HashedList`. The `HashedList` interface from the stdlib provides the hash table machinery — buckets, linear probing, rehash at 75% load, tombstone removal. The `hash_key` method on `DictEntry` delegates to the key's `get_hash`.

The methods are external functions with where clauses:

```
pub func Dict.set<K, V>(self, key: K, value: V) where K: Hashable
    let entry = DictEntry<K, V> { key: key, value: value }
    hash_insert<Dict<K, V>, DictEntry<K, V>>(self, entry)
}

pub func Dict.get<K, V>(self, key: K) -> DictEntry<K, V>? where K:
```

```
    let h = key.get_hash()
    return hash_lookup<Dict<K, V>, DictEntry<K, V>>(self, h)
}
```

`.set()` creates a `DictEntry` and calls `hash_insert` — the same function that powers `SymTable`. `.get()` computes the hash and calls `hash_lookup`. The generic parameters `<K, V>` flow through monomorphization: `Dict<Sym, i32>` generates specialized C functions with `Sym` and `int32_t` baked in. The Lyric compiler itself uses `Dict<Sym, TypeInfo>` for its symbol table, `Dict<Sym, LFuncDecl>` for the lowerer, and `Dict<Sym, string>` for class renames — all from this same 30-line definition.

This is the payoff of the interface/relation system. `HashedList` is written once — 200 lines of Lyric handling buckets, probing, rehashing, and removal. `Dict` is 30 lines that wire it to a key-value pair. `SymTable` is 10 lines that wire it to interned strings. Neither duplicates any hash table logic.

0.53.6 10.5 A Symbol Table for Variable Bindings

Hash tables are the workhorse of every interpreter, compiler, and config loader. Here's a self-contained example: a tiny set of variable bindings that maps names to values, looks them up, and reports an `error` when a name isn't bound.

```
func get_var(d: Dict<Sym, f64>, name: string) -> (f64, error) {
    let entry = d.get(sym(name))
    if isnull(entry) {
        return (0.0, Error { msg: f"undefined variable: {name}" })
    }
    return (entry!.value, null)
}

func main() {
    let vars = Dict<Sym, f64>()
    vars.set(sym("pi"), 3.14159)
    vars.set(sym("e"), 2.71828)

    let (vpi, e1) = get_var(vars, "pi")
    if e1 == null {
```

```
        println(f"pi = {vpi}")
    }

    let (_, e2) = get_var(vars, "tau")
    if e2 != null {
        println(f"{e2}")
    }
}
```

Output:

```
pi = 3.14159
undefined variable: tau
```

The `Dict<Sym, f64>` holds the bindings; `sym(name)` interns the lookup key on the way in. If the name was already interned (a previous assignment, a previous lookup), `sym` returns the cached `Sym` — an O(1) hash table hit. If it's the first time, it interns the string. Either way, the `Dict` lookup uses the pre-computed `u64` hash, not the raw bytes.

The error path uses the same `Error { msg: ... }` literal and `(T, error)` tuple-return shape from Chapter 5, with `f"{e2}"` to stringify the error value. No `new_error` shortcut, no `.message()` call — just the idioms the language already has.

A natural next step is to wrap the `Dict<Sym, f64>` inside a `Calculator` or `VarTable` class so the bindings travel with the rest of the evaluator state. That doesn't compile today: declaring `vars: Dict<Sym, f64>` as a field on a non-generic class trips a `TypeVar leak 'V'` in the checker when a method calls `self.vars.get(...)` — the concrete `V = f64` from the field type isn't being propagated into `Dict.get`'s where-clause typevar. Making the outer class generic trips a different downstream monomorphization failure. Until both are fixed, the working pedagogy is a top-level `let vars = Dict<Sym, f64>()` plus free functions that take the dict as a parameter (as above).

None of this uses garbage collection. The `Dict` is a hash table built on relations. When `main` returns, `vars` goes out of scope, its `HashedList`-injected destructor walks the children and frees every `DictEntry`, and the slabs reclaim the slots. The intern table — the `permanent SymTable` from §10.1 — lives for the lifetime of the pro-

cess, which is what you want: a `Sym` value is a stable handle that never goes stale.

The next chapter looks at how all of this maps to memory — where structs live, where classes live, where slabs live, and what `permanent` actually does to a class.

0.54 Chapter 11: Memory Management — No GC, No Borrow Checker

The previous chapters showed how to build data structures, declare ownership with relations, and wire interfaces. All of that produces programs that never call `free`. No garbage collector runs. No borrow checker rejects your code. This chapter explains what actually happens underneath.

0.54.1 11.1 The Three Memory Regions

Lyric has three kinds of values, and each lives in a different place:

Structs live on the stack. They're value types — copied on assignment, passed by value, freed when the enclosing scope exits. A struct with three `i32` fields occupies 12 bytes on the stack frame, no heap allocation, no indirection.

Classes live on the heap, allocated from typed slab allocators. Every class type gets its own slab. `Node { name: "root" }` doesn't call `malloc` — it grabs the next slot from the `Node` slab. Class variables hold pointers (in AoS mode) or integer handles (in SoA mode). Assignment copies the pointer, not the object. Two variables can refer to the same class instance.

Slices are fat pointers — a `(data, len, cap)` triple. The backing array is heap-allocated and shared on assignment. `let b = a` makes `b` point to the same array as `a`. This is the same model as Go slices.

Here's how this plays out:

```
struct Point {
    x: f64
    y: f64
}
```

```
class Particle {
    pos: Point
    name: string
}

func main() {
    let p1 = Point { x: 1.0, y: 2.0 }
    let p2 = p1   // copy - p2 is independent

    let a = Particle { pos: p1, name: "alpha" }
    let b = a     // b points to the same Particle as a

    let mut items: [i32] = []
    items.push(1)
    let items2 = items  // items2 shares the backing array
}
```

Modifying `p2.x` does not affect `p1.x` — they're separate stack values. But `a` and `b` are the same particle; changing `a.name` changes `b.name` too. And `items` and `items2` share the same backing array — at least until one of them calls `push` and triggers a reallocation.

This is the value-type lesson from Chapter 2, extended to the full memory model. Structs copy. Classes share. Slices share the backing array.

0.54.2 11.2 Slab Allocation

Every class type gets a typed slab allocator. When you write `Node { name: "root" }`, the compiler emits a call to `_lyric_slab_alloc_Node()`. Here's the generated C for a simple `Node` class:

```
/* Slab allocator infrastructure (AoS block-based) */
typedef struct LyricSlab_Node_Block {
    struct Node data[LYRIC_SLAB_BLOCK];
    struct LyricSlab_Node_Block* next;
    int32_t used;
} LyricSlab_Node_Block;
typedef struct { LyricSlab_Node_Block* cur; Node* free; } LyricSla
static LyricSlab_Node _lyric_slab_Node = {0};
```

```
static Node* _lyric_slab_alloc_Node(void) {
    if (_lyric_slab_Node.free) {
        Node* p = _lyric_slab_Node.free;
        _lyric_slab_Node.free = p->lyric_next;
        memset(p, 0, sizeof(Node));
        return p;
    }
    if (!_lyric_slab_Node.cur ||
        _lyric_slab_Node.cur->used == LYRIC_SLAB_BLOCK) {
        LyricSlab_Node_Block* b = calloc(1, sizeof(*b));
        b->next = _lyric_slab_Node.cur;
        _lyric_slab_Node.cur = b;
    }
    return &_lyric_slab_Node.cur->data[_lyric_slab_Node.cur->used+
}
```

Allocations come from a contiguous block of 256 objects (`LYRIC_SLAB_BLOCK`). When a block fills, a new one is allocated. When an object is freed, it goes on a free list threaded through a `lyric_next` field that the compiler adds to every class. The next allocation reuses that slot.

This is the default: Array-of-Structs (AoS) layout. Each `Node` is a contiguous chunk of memory — name, children, lyric_next — stored together. This is the natural layout that C programmers expect.

0.54.3 11.3 The –soa Flag

Compile the same program with `--soa` and the generated C changes fundamentally:

```
/* Slab allocator infrastructure (SoA parallel-array) */
typedef struct {
    lyric_string* name;
    LyricSlice_Node* children;
    uint32_t* lyric_next;
    uint32_t used;
    uint32_t cap;
    uint32_t free_head;
} LyricSlab_Node;
```

```
static LyricSlab_Node _lyric_slab_Node = { .used = 1 };
```

Instead of an array of `Node` objects, there are parallel arrays — one per field. All the `name` strings are contiguous in memory. All the `children` slices are contiguous. All the `lyric_next` pointers are contiguous.

Class handles change from `Node*` pointers to `uint32_t` indices. `Node { name: "root" }` returns an integer handle; field access becomes `_lyric_slab_Node.name[h]` instead of `p->name`.

Why does this matter? Cache lines. A modern CPU loads 64 bytes at a time. In AoS layout, loading a `Node`'s name pulls in the children, the lyric_next, and padding — wasting cache space on fields you don't need. In SoA layout, iterating over all names touches only the name array. Every cache line is full of names.

The Lyric compiler itself benchmarks at **10% faster and 14% less memory** under SoA compared to AoS, measured by compiling the compiler's own 33,500-line codebase on a MacBook Air M2. The program doesn't change — same source code, same semantics. Only the `--soa` flag changes.

We proved this at scale with DataDraw over 30 years: EDA tools processing billions of transistor records, where cache-line utilization determined whether a job finished in minutes or hours. Lyric brings the same technique to a general-purpose language, and you don't have to redesign your data structures to get it.

0.54.4 11.4 Deterministic Destruction

Classes are freed through relations. When a parent with an `owns` relation is destroyed, the destruction cascades to all children:

```
class TeamA { name: string }
class TeamB { name: string }
class Player { name: string }

relation OwningList TeamA:team_a owns [Player:pa]
relation OwningList TeamB:team_b owns [Player:pb]

func main() {
    let a = TeamA { name: "Alphas" }
```

```
    let b = TeamB { name: "Betas" }
    let p = Player { name: "Alice" }

    a.team_a_append(p)
    b.team_b_append(p)

    println(f"a has player: {!isnull(a.team_a_first)}")
    println(f"b has player: {!isnull(b.team_b_first)}")

    let old_ptr = p

    a.destroy()

    println(f"b has player after destroy: {!isnull(b.team_b_first)

    let p2 = Player { name: "Bob" }
    println(f"slab reuse: {p2 == old_ptr}")
    println(f"p2 name: {p2.name}")
}
```

Output:

```
a has player: true
b has player: true
b has player after destroy: false
slab reuse: true
p2 name: Bob
```

When `a.destroy()` fires, it cascade-destroys Alice (because `TeamA` *owns* her). Alice is removed from both `TeamA`'s list and `TeamB`'s list. Then her slab slot goes on the free list — `memset` zeros the slot, so any dangling reference sees zeroed fields rather than garbage. The next allocation (`Player { name: "Bob" }`) reuses that same slot.

After `a.destroy()`, `p` is a dangling pointer. Accessing `p.name` is undefined behavior, even though the zeroed memory makes it look safe. The slab allocator's `memset` is a debugging aid, not a safety guarantee. Don't rely on it.

This is the one place Lyric's safety story has a real gap today: a stale reference to a destroyed object is a use-after-free. The roadmap fix is ***bidirectional pointers*** *— a back-pointer annotation that*

the compiler tracks across destroys, automatically nulling it when the owner dies. Combined with a planned `destroys` *annotation (declares "this function may destroy* `self`*") and* `mut resize` *(declares "this function may reallocate the backing array"), the checker will be able to reject UAF at compile time. Until then: when you have references that outlive an* `owns` *relation, keep them inside* `if !isnull(parent)` *guards, or hold them through the parent (*`team.roster_children[i]`*) rather than as standalone pointers. While debugging, compile with* `--detect-uaf` *— freed slab slots are marked with a sentinel and every class access checks for it, turning silent UAF into a loud crash at the point of the bad read.*

A class can participate in multiple `owns` relations simultaneously — Alice was owned by both TeamA and TeamB. Whichever owner's `destroy` fires first cascade-destroys the child. The child's destructor automatically unlinks it from all other relations before the slab slot is freed.

This is deterministic. No GC pause, no finalization queue, no reference cycle detection. The ownership graph declared by relations is the destruction order. The compiler generates the cascade code — you never write a destructor. Every non-`permanent` class gets a `destroy(self)` method synthesized for it automatically, with a body assembled from all the relation destructors and any `final func` cleanup you declared (Chapter 8 §8.7 introduced `final`). The default body for a class with no relations and no `final` just frees the slab slot.

0.54.5 11.5 The Three Lifetime Regimes

The previous section showed `owns` driving destruction, but `owns` is one of three regimes the compiler picks per class. The spec spells them out; here's the working version:

1. **Owned.** The class is the child of an `owns` relation. Its lifetime is the parent's lifetime. No reference count is maintained — the parent's cascade is the only way it dies. This is the regime every example so far has used: `Player` owned by `Team`, `DictEntry` owned by `Dict`, `Stmt` owned by `Program`.
2. **Permanent.** The class is declared `permanent class Foo`. Instances are never freed and never reference-counted. The slab

grows; nothing ever returns to the free list. Chapter 10's `SymTable` is the canonical example — interned `Sym` values must outlive every function that holds a handle, so the entire intern table opts out of reclamation. Use `permanent` for compiler singletons and for AST or symbol trees that have whole-program lifetimes. *Roadmap: a `permanent` class that is also a relation target will produce a compile-time warning, since the two policies contradict.*

3. **Refcounted.** Every other class — anything that's neither owned by a relation nor declared `permanent`. The compiler inserts `ref` increments at assignment and `ref` decrements at scope exit, and destroys the instance when the count hits zero. This is what gives local class values their "Go-like" feel: you create one, pass it around, and it goes away when the last variable referring to it disappears. The `--rc-free` flag (on by default) is what wires "RC = 0" to `destroy()`; `--no-rc` disables auto-destruction for benchmarks that want to measure the cost separately.

Two compiler details worth knowing because they show up in the generated C:

- **Move inference, not move syntax.** If you assign a local class variable into a struct field or pass it to a function, and you never touch that local again afterward, the lowerer treats the assignment as a *move* — no retain/release pair is emitted around it. You never write `move x` or anything like Rust's `&T` vs `T`; dataflow analysis figures it out. The effect is invisible at the Lyric level, but it's why you'll see fewer RC operations in the generated C than you might expect.
- **`ref` and `unref` are escape hatches.** The stdlib's `ArrayList`, `OwningList`, and `RefList` implementations call `ref child` and `unref child` directly — bare RC primitives that are only legal inside a `trusted` function. You won't write these unless you're building your own ownership container.

0.54.6 11.6 Scope-Exit Analysis

Not every class participates in a relation. The compiler also runs escape analysis to free locally-created values at scope exit:

```
struct Holder { items: [i32] }

func test_local_no_escape() {
    let mut temps: [i32] = []
    temps.push(1)
    temps.push(2)
    let mut sum = 0
    let mut i = 0
    while i < temps.len() {
        sum = sum + temps[i]
        i = i + 1
    }
    assert_eq(sum, 3, "local no escape")
    // temps is freed here - it never escaped this scope
}

func make_list() -> [i32] {
    let mut result: [i32] = []
    result.push(10)
    result.push(20)
    return result
    // result is NOT freed - it's returned to the caller
}

func test_struct_field_escape() {
    let mut items: [i32] = []
    items.push(42)
    let h = Holder { items: items }
    // items is NOT freed - it escaped into the struct field
}
```

The escape analysis runs at fixed-point iteration. First pass: mark parameters that get stored into struct or class fields. Second pass: mark parameters passed to another function's escaping parameter position. Repeat until no changes. Any slice created locally (via `[]` literal or `push`/`append`) that doesn't escape — isn't returned, isn't stored in a field, isn't passed to an escaping parameter — gets a free call injected at scope exit.

The same analysis applies to strings created by f-strings and concatenation, and to class instances allocated locally that aren't part of an

`owns` relation.

The analysis is conservative. If a slice is assigned to another variable (`let b = a`), both are marked as potentially escaping — the compiler doesn't track which one is the "owner" after assignment. This is the same trade-off Go makes with slice aliasing. Correctness over optimization.

If a lambda captures a local slice, the slice is marked as escaping — the lambda might outlive the current scope. The same applies to `spawn` blocks, which capture variables by pointer (see Chapter 12).

0.54.7 11.7 Copy-on-Assign: The Recurring Lesson

The value-type model has one consistent gotcha. When you modify a struct and forget it's a copy, the modification is lost:

```
struct Config {
    debug: bool
    verbose: bool
}

func enable_debug(c: Config) {
    // c is a copy - this modification is lost when the function r
    // To modify the original, use: func enable_debug(mut c: Confi
}
```

Without `mut`, the function receives a copy. The fix is always `mut` — pass by mutable reference so the caller sees the change. This applies everywhere structs appear — function parameters, slice indexing (which returns a copy), optional unwrapping (which returns a copy).

The same principle doesn't apply to classes. Classes are pointers. When you pass a class to a function, the function sees the same object. Mutations are visible to the caller. No `mut` needed — and in fact, using `mut` on a class parameter creates a double-pointer, which is almost never what you want.

0.54.8 11.8 What the Calculator Costs

The calculator from Chapters 4–5 is the largest program the book has built. Let's count what it allocates:

- **`Token` and `TokenKind`** — both stack values. A `Token` is a struct holding a `TokenKind` enum and a `string` field. The `string` is a fat-pointer view into the original source — no copy. Zero heap cost for the tokens themselves; one shared backing buffer for the source text.
- **`[Token]` token list** — one slice allocation, freed at scope exit by the escape analysis described in §11.6. The slice doesn't survive past `parse`.
- **`Parser`** — one slab allocation. It holds the token slice and a cursor. Refcounted (no `owns` relation, not `permanent`); freed when the last reference drops, which in the current `parse` shape is the end of the function.
- **No `malloc`, no `free`.** Everything class-shaped lands in a typed slab; everything slice-shaped is reclaimed by scope-exit analysis.

If you bolt the §10.5 variable-bindings example onto the calculator, the inventory grows by exactly two more line items:

- **`Dict<Sym, f64>`** and its **`DictEntry<Sym, f64>`** children — one `Dict` slab slot plus one `DictEntry` slab slot per defined variable. The `Dict` `owns` its entries via a `HashedList` relation; destroying the `Dict` cascade-destroys every entry. No leak path exists.
- **`Sym`** instances — interned in the global `permanent SymTable` (Chapter 10 §10.1, §11.5). Created once per unique variable name and never freed. This is exactly what `permanent` is for.

Compile with `--soa` and the memory footprint shrinks again: each `DictEntry` field becomes a column in a parallel array instead of a row in an AoS struct, and class handles become 32-bit indices instead of 64-bit pointers. The program runs the same — same source code, same output. Only the layout changes underneath.

This is what Lyric's memory model delivers: you write ownership declarations, and the compiler generates an allocation strategy that would take hundreds of lines of C to implement manually. No garbage collector. No borrow checker. No unsafe blocks. Just declarations and generated code.

0.55 Chapter 12: Concurrency

```
func main() {
    let done = make_channel<bool>(1)

    spawn {
        let x = 42
        println(f"hello from goroutine: {x}")
        done.send(true)
    }
    done.receive()

    println("all done")
}
```

`spawn` launches a block on a new thread. `make_channel<T>()` creates a typed channel. `send` and `receive` are methods on the channel. That's the entire concurrency model. If you've used Go, this is familiar — goroutines and channels, with method syntax instead of arrow operators.

0.55.1 12.1 Spawn

`spawn` takes a block and runs it concurrently:

```
func main() {
    let done = make_channel<bool>(1)

    spawn {
        for i in [1, 2, 3] {
            println(f"item: {i}")
        }
        done.send(true)
    }
    done.receive()

    spawn {
        println("third goroutine")
        done.send(true)
    }
    done.receive()
```

```
    println("all done")
}
```

Variables from the enclosing scope are captured automatically. You don't declare what to capture — the compiler walks the block, finds every name that resolves to the enclosing scope, generates a context struct with those fields, and passes a pointer to it when launching the thread. Each `spawn` becomes a `pthread_create` call with an auto-generated wrapper. There's no green-thread runtime and no thread pool — one `spawn`, one OS thread. The generated C looks roughly like this:

```
typedef struct {
    lyric_string* x;
    LyricChan_bool* done;
} _spawn_1_ctx;

void* _spawn_1(void* _arg) {
    _spawn_1_ctx* _ctx = (_spawn_1_ctx*)_arg;
    // ... body using _ctx->x, _ctx->done ...
    free(_ctx);
    return NULL;
}
```

`spawn` captures by pointer, which is a data race waiting to happen. Both the parent and the spawned block see the same memory for every captured variable, so a write on either side races with a read or write on the other. Channels are safe to capture this way because they're already class pointers with internal locking, but a captured `let mut counter: i32 = 0` mutated by two `spawn` blocks is a textbook race — no warning, no help. The roadmap intent is copy-by-value capture with explicit shared mutation through channels or locks. Until that lands: if you need isolation, copy the value into a local *inside* the spawned block (`let local = captured`) before mutating; for genuinely shared mutable state, use `lock` (§12.5) or — better — funnel updates through a channel and let one owner do the writes.

0.55.2 12.2 Channels

Channels are typed, first-class values. Create them with `make_channel<T>()` for unbuffered or `make_channel<T>(n)` for a buffer of size `n`:

```
func main() {
    // Buffered channel - holds up to 10 values
    let ch = make_channel<i32>(10)

    ch.send(42)
    let val = ch.receive()
    println(f"received: {val}")

    // Unbuffered channel - send blocks until someone receives
    let done = make_channel<bool>()

    spawn {
        let ch2 = make_channel<string>(1)
        ch2.send("hello")
        let msg = ch2.receive()
        println(msg)
        done.send(true)
    }

    done.receive()
    println("all done")
}
```

Three methods: `send(value)`, `receive()`, and `close()`. An unbuffered channel blocks the sender until a receiver is ready, and vice versa. A buffered channel blocks only when the buffer is full.

The C backend generates a typed channel struct for each element type used in the program — `LyricChan_i32`, `LyricChan_string`, `LyricChan_bool`. Each contains a pthread mutex, condition variables, and a circular buffer. The monomorphizer specializes the channel implementation per type, just like it does for generic functions. No `void*` casting, no type erasure.

0.55.3 12.3 The Producer Pattern

Channels and spawn combine naturally into producer-consumer patterns:

```
func producer(ch: channel<i32>, count: i32) {
    let mut i: i32 = 1
    while i <= count {
        ch.send(i)
        i = i + 1
    }
    ch.close()
}

func main() {
    let ch = make_channel<i32>(10)
    spawn {
        producer(ch, 5)
    }

    let mut val = ch.receive()
    while val > 0 {
        println(f"got: {val}")
        val = ch.receive()
    }
    println("producer done")
}
```

The producer sends `1, 2, 3, 4, 5` and closes the channel. The consumer receives until it gets a zero — which is what a closed channel returns for `i32`. Notice the deliberate choice to start counting from `1`: we need a real sentinel.

`receive()` on a closed channel returns the zero value for the type (`0` for integers, `""` for strings, `false` for `bool`) with no indication that the channel closed. There's no `(value, ok)` tuple like Go's `v, ok := <-ch`. The roadmap target is a `(T, bool)` form: `let (v, ok) = ch.receive()`. Until that lands, you have three options: (1) pick a domain value that the producer will never send and use it as a sentinel (what this example does — `0` is the sentinel because we start sending from `1`); (2) use a separate `done` channel to signal completion; (3) wrap your real values in an `Optional` and treat `null`

as the close signal. Option 2 is the most robust when you don't control the producer's value range.

Channels are passed by reference — the spawned block and the main function share the same channel object, which is exactly what you want. Channels carry their own internal mutex and condition variables, so capturing a channel by pointer (§12.1) is safe; concurrent `send`s and `receive`s serialize correctly.

0.55.4 12.4 Select

When you need to wait on multiple channels, `select` picks whichever is ready first:

```
func main() {
    let ch1 = make_channel<string>(1)
    let ch2 = make_channel<i32>(1)

    ch1.send("hello")

    select {
        case msg = ch1.receive() => {
            println(f"got message: {msg}")
        }
        case num = ch2.receive() => {
            println(f"got number: {num}")
        }
    }
}
```

Each `case` binds a variable to the received value. If multiple channels are ready, one is chosen. If none are ready, `select` blocks until one becomes available.

You can also use `select` with send cases and a `default` branch:

```
func main() {
    let ch1 = make_channel<string>(1)
    let ch2 = make_channel<i32>(1)
    let done = make_channel<bool>(1)

    ch2.send(42)
```

```
    select {
        case val = ch2.receive() => {
            println(f"received: {val}")
        }
        default => {
            println("no data ready")
        }
    }

    spawn {
        let x = ch1.receive()
        println(f"spawned got: {x}")
        done.send(true)
    }

    select {
        case ch1.send("world") => {
            println("sent to ch1")
        }
    }

    done.receive()
    println("select done")
}
```

The `default` branch runs immediately if no channel is ready — turning a blocking select into a non-blocking poll. Send cases (`case ch.send(val) =>`) succeed when the channel has buffer space or a receiver is waiting.

`select` is not a true blocking primitive today. The C backend compiles it into a polling loop: try each case in turn, run `default` if present and no case is ready, otherwise `usleep(100)` and retry. Each case becomes a non-blocking `tryrecv` or `trysend` call on the underlying channel. This burns CPU on hot selects and adds ~100µs of latency on cold ones — the roadmap target is condvar / epoll-based wake. Until that lands, avoid tight select loops in latency-sensitive code, and prefer a dedicated channel-per-source design where possible.

0.55.5 12.5 Locks

For shared mutable state that doesn't fit the channel model, Lyric provides scoped mutexes:

```
func main() {
    let mut mu: lock
    let mut count: i32 = 0

    lock(mu) {
        count = count + 1
    }
    lock(mu) {
        count = count + 10
    }

    println(f"final count: {count}")
}
```

Output: `final count: 11`.

`lock` is a built-in type that zero-initializes — `let mut mu: lock` is valid without a constructor call. The C backend generates `pthread_mutex_t` with `PTHREAD_MUTEX_INITIALIZER`. `lock(mu) { ... }` acquires the mutex, runs the block, and releases it — even if the block returns early. In C, this compiles to `pthread_mutex_lock` and `pthread_mutex_unlock` bracketing the block body. The scoped syntax makes it impossible to forget the unlock.

Lowercase `lock` is the only spelling that compiles today. Older drafts (and some testdata files that haven't been updated yet) use `Mutex` or capital `Lock`; both have been removed and now fail with `unresolved type var 'Lock'`. If you're reading code from before mid-2026, expect to mechanically rename it.

0.55.6 12.6 Guarded Fields

Locks protect critical sections, but nothing stops you from accessing a guarded variable outside the lock. The `guarded_by` annotation fixes this:

```
class Counter {
    count: i32 guarded_by(mu)
```

```
    mu: lock
    label: string

    pub func increment(self) {
        lock(self.mu) {
            self.count = self.count + 1
        }
    }

    pub func get_label(self) -> string {
        return self.label
    }
}

func main() {
    let c = Counter {}

    lock(c.mu) {
        let val = c.count
        println(val)
    }

    println(c.get_label())
}
```

The `count` field is annotated `guarded_by(mu)`. *Today the annotation parses and is stored on the field, but the checker does not enforce it — accessing `c.count` outside a `lock(c.mu)` block compiles cleanly. The design intent is what's described here: a compile-time error on un-guarded access. The roadmap item is to add the cross-function check.* The `label` field has no annotation — it'll always be accessible anywhere, even after the check is added. `guarded_by` is meant to be statically verifiable with no runtime overhead — just the basic discipline that a field should only be touched while its mutex is held.

Note that `guarded_by` is a contextual keyword — the lexer emits it as an identifier, and the parser recognizes it by context. This keeps the keyword list small and avoids breaking code that uses `guarded_by` as a variable name (unlikely, but possible).

0.55.7 12.7 Putting It Together

Channels, spawn, select, and locks compose naturally. Here's a concurrent accumulator — two spawned workers each compute a partial sum and ship it back through a channel, and the main function collects and combines:

```
func main() {
    let results = make_channel<i32>(10)
    let done = make_channel<bool>()

    // Two workers, each computing a partial sum
    spawn {
        let mut sum: i32 = 0
        for i in [1, 2, 3] {
            sum = sum + i
        }
        results.send(sum)
        done.send(true)
    }

    spawn {
        let mut sum: i32 = 0
        for i in [4, 5, 6] {
            sum = sum + i
        }
        results.send(sum)
        done.send(true)
    }

    done.receive()
    done.receive()

    let a = results.receive()
    let b = results.receive()
    println(f"total: {a + b}")
}
```

No shared mutable state. No locks. Each worker computes independently and sends its result through a channel. The main function collects and combines. This is the concurrency pattern Lyric encour-

ages — share memory by communicating, not by locking.

0.56 Chapter 13: Modules and Packages

For twelve chapters the calculator has lived in a single `calc.ly`. Real programs don't. This chapter is about how Lyric organizes code across files and directories — packages, modules, imports, the standard library — and about what's deliberately *missing* from that story today.

The model is intentionally simple. A package is a directory. A module is a project. Visibility is `pub` or private. There's no separate compilation, no link step, no header files, no package registry. The whole program — your code plus the stdlib plus every imported package — is parsed, merged, type-checked, and emitted as a single C file. At 33,000-line scale (the compiler itself), that compiles in under a second; we'll add incremental compilation when it stops scaling.

0.56.1 13.1 Packages, Imports, and `pub`

Here's a project with two packages:

```
mylib_demo/
   lyric.mod
   main.ly
   mylib/
       types.ly
       utils.ly
```

`mylib/types.ly`:

```
lyric mylib {
    pub struct Point {
        x: i32
        y: i32
    }

    pub func new_point(x: i32, y: i32) -> Point {
        return Point { x: x, y: y }
    }
}
```

`mylib/utils.ly`:

```
lyric mylib {
    pub func add(a: i32, b: i32) -> i32 {
        return a + b
    }
}
```

And a main file that uses it:

```
lyric main {
    import mylib

    func main() {
        let p = mylib.new_point(1, 2)
        let sum = mylib.add(p.x, p.y)
        println(f"{sum}")
    }
}
```

The fourth file, `lyric.mod`, is one line: `module mylib_demo`. It marks the project root. Compile and run:

```
$ lyric compile mylib_demo -o mylib_demo.c
$ gcc -std=gnu11 -O2 -w -I runtime -o mylib_demo mylib_demo.c -lm
$ ./mylib_demo
3
```

Four things happened. First, each file wraps its declarations in `lyric mylib { }` — that's the package declaration. The block name is conventional; the *real* package name is the directory's name. All `.ly` files in `mylib/` belong to package `mylib`, and `Point` defined in `types.ly` is visible in `utils.ly` without any import.

Second, `pub` controls visibility across packages. Without `pub`, a function or type is package-private — the same keyword you've used throughout the book, now with a reason to exist. (***`pub` isn't actually filtered yet — see §13.7. Write it anyway so your code is correct when the filter lands.***)

Third, `import mylib` in `main.ly` makes `pub` *functions* in `mylib` accessible with the `mylib.` prefix: `mylib.new_point(...)`, `mylib.add(...)`. The compiler finds the `mylib/` directory by name, relative to the module root. (Qualifying *types* — `let p:`

`mylib.Point = ...` or `mylib.Point { x: 1, y: 2 }` — has sharp edges today; §13.5 walks through what works and what doesn't.)

Fourth, `lyric compile mylib_demo` (passing the directory, not a file list) tells the compiler to look for `lyric.mod` and collect every top-level `.ly` file in that directory. Imports in the root file then pull in subdirectory packages by name.

For single-file programs the `lyric name { }` wrapper is optional — bare top-level declarations belong to an implicit package derived from the filename. For multi-file projects, it's how you organize code.

0.56.2 13.2 How It Actually Works

The module system operates at the AST level, not through linkers or object files. When the compiler sees `import mylib`, it:

1. Looks for a `mylib/` directory under the module root
2. Parses every `.ly` file in it
3. Prefixes every top-level declaration name with `mylib_` — so `Point` becomes `mylib_Point`, `new_point` becomes `mylib_new_point`
4. Rewrites qualified access (`mylib.new_point`) to the prefixed name (`mylib_new_point`)
5. Merges the prefixed declarations into the program's flat namespace

That's it. No separate compilation, no linking, no symbol tables. The entire program — your code plus all imported packages plus the stdlib — becomes one compilation unit. The C backend emits one `.c` file containing everything.

This is deliberately simple. The Lyric compiler itself is 33,500 lines of Lyric across 14 files in 12 directories (`ast`, `lexer`, `parser`, `checker`, `desugar`, `lowerer`, `lir`, `optimizer`, `monomorphizer`, `c_backend`, `memory`, `main`). It all merges into a single 114,770-line C file. The whole pipeline — parse, check, lower, optimize, monomorphize, emit — runs in about 0.2 seconds. Separate compilation is an optimization you add when build times matter; at this scale, we haven't needed it.

0.56.3 13.3 The Module File

A `lyric.mod` file marks a project root:

```
module calculator
```

That's the entire file — one line declaring the module name. When you run `lyric compile .` (passing a directory rather than a file list), the compiler looks for `lyric.mod` in that directory. If it finds one, it collects every top-level `.ly` file in the directory, resolves the imports those files make, and compiles everything together.

The `lyric.mod` file serves the same purpose as Go's `go.mod` or Rust's `Cargo.toml`: it tells the toolchain where your project starts. Unlike those files, today it has no dependency management, no version constraints, no build configuration. Lyric doesn't have a package registry yet. If you need external code, copy it into your source tree.

*Roadmap: today the compiler only checks for `lyric.mod`'s **existence** — its contents aren't parsed. The intent is for `lyric.mod` to declare the module's import-path prefix, its external dependencies, and the package containing `main()`. Until that parsing lands, drop a one-line `module name` (the name is for humans; the compiler ignores it) and rely on the directory layout.*

0.56.4 13.4 The Standard Library

You've been using `println`, `append`, `assert_eq`, `Dict`, `ArrayList`, and dozens of other functions throughout this book without ever writing `import std`. The standard library is auto-imported — the compiler merges it into your program before type checking, without any explicit import.

The stdlib is two files totaling 991 lines of Lyric:

- **`std.ly`** (733 lines): `ArrayList`, `OwningList`, `RefList`, `HashedList`, `Dict`, `Sym`, `StringBuilder`, `Error` — all the interfaces, relations, and data structures from Chapters 8–10.
- **`string.ly`** (258 lines): string methods — `split`, `trim`, `contains`, `index_of`, `replace`, `has_prefix`, `has_suffix`, `to_upper`, `to_lower`, `join`, and the rest.

Every line is Lyric. No C escape hatches, no compiler magic. When you call `dict.set(key, value)`, you're calling a Lyric method de-

fined in `std.ly` using the same interfaces and relations this book taught you. The stdlib is the proof that Lyric's features compose into real libraries.

The merge is not blind: a pass called `merge_stdlib` walks your code, sees which stdlib types and functions you actually reference, and pulls in just those (plus their transitive dependencies) to a fixed point. So `Dict` literals always pull in `Dict`, but a program that never touches `StringBuilder` doesn't pay for it in the emitted C.

0.56.5 13.5 Splitting the Calculator

The calculator we built in Chapters 4 and 5 has grown large enough that one file is starting to feel cramped. There are two ways to split it: across multiple files in the same package, or across packages with `import`. The first is unconditional and ergonomic; the second has sharp edges today. Let's do the first, and the next section will be honest about the second.

Here's a three-file split, all in package `main`:

```
calculator/
   lyric.mod
   lexer.ly        // tokenize() - package "main"
   parser.ly       // parse() - package "main"
   main.ly         // main() - package "main"
```

`lexer.ly` declares the tokenizer:

```
lyric main {
    pub enum TokenKind {
        Number
        Plus
        Minus
        Star
        Slash
        LeftParen
        RightParen
        End
    }

    pub struct Token {
```

```
        kind: TokenKind
        value: string
    }

    pub func tokenize(input: string) -> [Token] {
        // ... the Chapter 4 tokenizer, unchanged ...
        let mut tokens: [Token] = []
        // (body elided for space - same code as §4.7)
        return tokens
    }
}
```

This packaging adds an `End` variant to `TokenKind` for completeness — handy when a parser wants to look one past the last token without bounds-checking. The `End` token isn't produced by Chapter 4's `tokenize` shown there; treat §13.5 as a fresh package-split example, not a re-export of the calculator's runtime types.

`parser.ly` uses `Token` and `TokenKind` directly — no `import`, no `lexer.` prefix:

```
lyric main {
    pub class Parser {
        tokens: [Token]
        pos: i32
    }

    pub func parse(input: string) -> (f64, error) {
        let toks = tokenize(input)
        let p = Parser { tokens: toks, pos: 0 }
        return p.parse_expr()
    }

    // ... parse_expr / parse_term / parse_primary, exactly as in
}
```

And `main.ly` ties them together:

```
lyric main {
    func main() {
        let (val, err) = parse("(3 + 4) * 2")
        if err != null {
```

```
            println(f"error: {err}")
        } else {
            println(f"= {val}")
        }
    }
}
```

The `lyric main { }` wrapper in each file is conventional — the compiler doesn't actually need it; the package name comes from the directory. Both files share one namespace, so `parser.ly` calling `tokenize(...)` resolves to the function declared in `lexer.ly` with zero ceremony. Compile and run with `lyric compile calculator`.

What this split *doesn't* give you is enforcement. Anything you wanted to keep private to the lexer (a helper like `is_digit`) is visible from `parser.ly` and `main.ly` just the same. To get a hard boundary you'd need a separate package — and that's where the sharp edges live.

True cross-package today: function calls only. Pull the lexer into its own package and the import system will resolve calls into it, but only the kind of call where the function returns a value you then use unqualified:

```
// lexer/lexer.ly
lyric lexer {
    pub struct Token { kind: i32, value: string }

    pub func make_number(value: string) -> Token {
        return Token { kind: 1, value: value }
    }

    pub func tokenize(input: string) -> [Token] {
        let seed = make_number("")
        let mut tokens = [seed]
        tokens.pop()
        // ... fill in tokens, return ...
        return tokens
    }
}

// main.ly
```

```
lyric main {
    import lexer

    func main() {
        let toks = lexer.tokenize("(3 + 4)")
        println(f"{toks.len()} tokens")
    }
}
```

`lexer.tokenize(...)` resolves; `toks[0].value` works (field access on a returned struct is fine). What does *not* work today:

- Naming the imported type in an annotation: `let xs: [lexer.Token] = []` — *checker rejects `lexer.Token` as an unknown type.*
- Constructing it at the call site: `lexer.Token { kind: 1, value: "x" }` — *same path, same rejection.*
- Referring to an imported enum variant: `lexer.TokenKind.Plus` — *same.*

The compiler accepts qualified function calls (`lexer.tokenize`) and qualified function references generally, but not qualified *type* names or *variant* names. The working pattern for now is to expose constructor functions (`make_number`, `new_point`) and let callers stay typed structurally — `let p = mylib.new_point(1, 2)` works because the type `Point` is inferred from `new_point`'s return type, never spelled at the call site.

*Roadmap: qualified type and enum-variant resolution — `let xs: [lexer.Token] = []`, `lexer.Token { ... }`, and `lexer.TokenKind.Plus` should all parse and check the same way `lexer.tokenize(...)` does. Until they do, package boundaries are best for **behaviour** (functions and methods) rather than **data** (types and variants). Keep types in the package that owns the algorithms over them.*

*Roadmap: recursive import resolution — only the root file's `import` statements are processed today. If `parser/parser.ly` says `import ast`, that import is silently ignored. Every package your program touches must be visible from `main.ly`'s imports, which in practice limits how deep package hierarchies go. The compiler itself sidesteps this by **not using `import` at all** — see §13.9.*

0.56.6 13.6 Import Variants

The parser accepts three forms of import:

```
import mylib                // by-name: resolves to ./mylib/, acces
import ml from "mylib"      // aliased: resolves to ./mylib/, acces
import "path/to/lib"        //   bare path - see below
```

The first form is what you'll use most. The second renames a package at the import site, useful when directory names are long or would collide with a local identifier. The path is interpreted relative to the module root (the directory holding `lyric.mod`).

Roadmap: bare `import "path"` parses but currently crashes the resolver because the alias derivation isn't implemented. Use `import alias from "path"` instead until the unaliased form is fixed.

0.56.7 13.7 What Packages Can't Do (Yet)

§13.5 named the two big constraints — single-level imports and call-only cross-package resolution. A few smaller things round out the picture:

`pub` isn't filtered across imports yet. *Every declaration in an imported package is visible after prefixing — package-private declarations leak. The roadmap target is true `pub` filtering: non-`pub` declarations should be invisible to importers. Write `pub` on everything you intend to export now, so your code is correct once the filter lands.*

No cycle detection. *Today there's no cycle detector — circular imports either work by accident or blow up with a duplicate-declaration error from the merge pass. The single-level rule makes the question mostly moot in practice; cycle detection becomes load-bearing once recursive resolution lands. The roadmap fix is the standard topological-sort error: "cycle detected: $a \to b \to c \to a$."*

No re-exports. If `parser` imports `ast`, the types and functions from `ast` don't become part of `parser`'s public API. Callers who need `ast.something` must import `ast` themselves.

No package registry. There's no `lyric get` or `lyric add`. If you want third-party code, copy it into your project. This is intentional for now — dependency management is a solved problem with

unsolved social problems (supply chain attacks, version conflicts, diamond dependencies). We'll add it when the language is mature enough to get it right.

One module, one compilation. Every import is resolved and merged at compile time. There are no pre-compiled libraries, no `.o` files, no dynamic linking. The entire program is one C file. The amortized-doubling `append` (like Go's, with O(1) amortized push) scales to 33,500 lines of Lyric in about 0.2 seconds. When that stops scaling, we'll add incremental compilation.

0.56.8 13.8 The `.lyric` Sibling

Every `.ly` file in this chapter is *implementation* code — function bodies, struct fields, the works. Lyric also has a declaration-only dialect, `.lyric`, that contains the same `pub` signatures and type declarations but no function bodies. A `.lyric` file is what other tooling reads when it wants to know your module's public surface without compiling its implementation.

The toolchain that reads `.lyric` files is called **lyre**. It's a separate program (written in Go, Python, TypeScript, and Lyric — one implementation per language ecosystem) that:

- Extracts `.lyric` from `.ly` (so the declaration file stays in sync with the implementation).
- Verifies that an implementation matches its declaration (signatures, fields, visibility).
- Hosts the contract-driven development annotations — `why:`, `doc`, `invariant:`, `source:`, `fake:` — that some Lyric projects use for design-by-contract.

Those CDD annotations live in `.lyric` files, not `.ly` files. The Lyric compiler doesn't see them, and the language reference doesn't mention them. They're a lyre feature; if you're not using lyre, you can ignore the `.lyric` mode entirely. The preface's "A sibling artifact: lyre" subsection has the full pitch.

0.56.9 13.9 The Compiler as Example

The Lyric compiler is the largest Lyric program in existence: 33,500 lines across 14 files in 12 directories. Its structure is a practical

demonstration of how to scale a Lyric project today:

```
src/
   ast/           ast.ly, modules.ly
   lexer/         lexer.ly
   parser/        parser.ly, expr_parser.ly
   checker/       checker.ly
   desugar/       desugar.ly
   lir/           lir.ly
   lowerer/       lowerer.ly
   optimizer/     optimizer.ly
   monomorphizer/ monomorphizer.ly
   memory/        memory.ly
   c_backend/     c_backend.ly
   main/          main.ly
```

Each directory is a package. `parser.ly` and `expr_parser.ly` both say `lyric parser { }` and share all declarations without imports — that's the multi-file-in-one-package shape from §13.5. What might surprise you: there is **not a single `import` statement** in the entire compiler. `parser.ly` calls `tokenize(...)` and constructs `Token { ... }` directly, even though those are declared in `lexer.ly` over in `src/lexer/`. The 14 files are simply listed together on the build command line, the parser merges them, and every declaration ends up in one flat namespace.

The Makefile makes this explicit:

```
BOOTSTRAP_FILES = \
  src/ast/ast.ly src/ast/modules.ly \
  src/lexer/lexer.ly \
  src/parser/parser.ly src/parser/expr_parser.ly \
  src/desugar/desugar.ly \
  src/checker/checker.ly \
  src/lir/lir.ly \
  src/lowerer/lowerer.ly \
  src/optimizer/optimizer.ly \
  src/monomorphizer/monomorphizer.ly \
  src/memory/memory.ly \
  src/c_backend/c_backend.ly \
  src/main/main.ly
```

```
update: lyric
    ./lyric compile $(BOOTSTRAP_FILES) -o lyric.c
```

Why this shape rather than `import`? Because §13.5's qualified-type limitation bites hardest in a compiler — the AST module wants to export *types* (`Expr`, `TokenKind`, `TypeInfo`), and qualified type names don't resolve across packages today. So instead of fighting the limitation, the compiler treats `lyric ast { }`, `lyric parser { }`, `lyric checker { }`, and so on as **logical sections** of a single program — directories give human-readable structure, `lyric name { }` blocks give a visual hint, and the merge pass treats it all as one namespace. *When qualified type resolution and recursive imports land, this code will be a candidate for a real* ***`import`*** *refactor.*

The whole thing compiles to one 114,770-line C file. `gcc` compiles that in a few seconds. The result is a binary that can compile itself — and the output matches byte-for-byte.

0.57 Chapter 14: The Self-Hosting Compiler

The Lyric compiler is written in Lyric. It parses Lyric source, type-checks it, lowers it to an intermediate representation, optimizes, monomorphizes generics, and emits C. Then GCC compiles the C. The whole process — 32,509 lines of Lyric across 14 files (plus 991 lines of stdlib) producing 114,770 lines of C — takes a few seconds end-to-end.

This chapter is a tour. Not a tutorial on how to write a compiler — that's a different book — but a walk through the pipeline that compiles every example in this one. By the end, you should understand how `./lyric compile` turns the `.ly` files you've been writing into a running binary, and why running it on its own source code reaches a fixed point.

0.57.1 14.1 The Pipeline

Here's `compile_pipeline` from `src/main/main.ly`, stripped to its skeleton:

```
func compile_pipeline(inputs: [string], output: string,
                      module_root: string, lir_dump: string,
                      soa: bool) -> bool {
```

```
// 1. Parse all input files
let mut all_files: [File?] = []
for input in inputs {
    let result = read_file(input)
    let parse_result = parse_file(result._0, input)
    all_files = append(all_files, parse_result._0)
}

// 2. Merge all files into one AST
let mut merged = merge_files(all_files)

// 3. Resolve module imports
if module_root != "" {
    let resolve_result = resolve_module_imports(module_root, m
    merged = resolve_result._0
}

// 4. Merge stdlib
let stdlib_dir = find_stdlib_dir()
if stdlib_dir != "" {
    let std_file = load_stdlib(stdlib_dir)
    if !isnull(std_file) {
        merge_stdlib(merged!, std_file!)
    }
}

// 5. Desugar
desugar_all(merged)

// 6. Type check
let checker = check_file(merged)

// 7. Lower to LIR
let lowerer = new_lowerer()
let prog = lowerer.lower_file(merged)

// 8. Optimize
optimize(prog!)  // dead code elimination, unused variable rem

// 9. Monomorphize
```

```
    monomorphize(prog)

    // 10. Validate post-monomorphization invariants
    validate_post_mono(prog)  // ensures no unresolved type params

    // 11. Rewrite impl renames to final names
    rewrite_impl_renames(prog)  // resolves label-prefixed method

    // 12. Slab allocation rewrite
    if soa { prog!.slab_mode_soa = true }
    slab_rewrite(prog!)

    // 13. Emit C
    let c_src = emit_c(prog)
    write_file(output, c_src)
    return true
}
```

Thirteen steps, one function, straight-line code. No pass manager, no plugin system, no visitor framework. Each step takes the output of the previous one and transforms it. The real code has error checks after each step — for instance, `parse_file` returns `(File?, error)` and uses `?` to propagate failures — but the skeleton shows the flow.

Let's walk through the interesting ones.

0.57.2 14.2 Parse

The parser (`src/parser/parser.ly`, 1,383 lines; `src/parser/expr_parser.ly`, 1,496 lines) is a recursive descent parser that produces an AST defined in `src/ast/ast.ly` (1,476 lines).

Splitting the parser into two files was practical, not architectural. Expression parsing is complex enough to deserve its own file — operator precedence, prefix/postfix, function calls, match expressions, if-expressions, lambdas. Both files declare `lyric parser { }` and share all declarations without imports.

One design choice worth noting: the parser uses a `no_struct_lit` flag to resolve an ambiguity. In `if x { ... }`, is `{ ... }` a struct literal or a block? Rust solves this by allowing struct literals everywhere except a few positions (conditions). Lyric takes the same

approach — when parsing the condition of `if`, `while`, `for`, or `match`, the parser sets `no_struct_lit = true`, which suppresses struct literal parsing. The braces are always a block in those positions.

0.57.3 14.3 The Middle Passes

Between parsing and C emission, the source passes through three major transformations:

Desugar (1,534 lines) runs six passes in fixed order: InterfaceEmbeds → InterfaceFields → FieldAccess → Relations → Destructors → DefaultImpls. The order is load-bearing — each pass generates AST nodes that later passes depend on. Relations (Chapter 8) and interfaces (Chapter 9) cover the design in detail; the key implementation insight is that destructor copies must be *deep* to prevent cross-relation contamination when method names are renamed.

Check (4,919 lines) is four-phase: Phase 0 pre-registers all type names so forward and cross-file references resolve; Phase 1 fills in the full `TypeInfo` (fields, methods, variants, type parameters, constraints); Phase 1.5 binds interface methods from impl blocks and where-clauses onto concrete classes, handling label-prefixed names; Phase 2 type-checks every function body and annotates every expression with its resolved type. Each phase must complete across ALL blocks before the next begins — this is what makes forward references and cross-file references work.

Lower (3,669 lines) translates the checked AST into LIR — a flattened, structured intermediate representation where `a + b * c` becomes `t1 = b * c; t2 = a + t1`. Control flow stays structured (if/while/match as statements, not basic blocks) because the C backend emits structured C. The lowerer also handles short-circuit `&&`/`||` (eager evaluation caused segfaults) and append write-back (without it, `append(obj.field, elem)` modifies a copy).

Optimize (1,556 lines) runs six LIR→LIR passes: fuse side-effect temps, destructure multi-returns, destructure extract-pairs from the `?` operator, fix nil-zero values on non-class returns, eliminate unused temps recursively while preserving side effects, and blank out unused multi-assign names. Each pass is small and local; together they undo the lowerer's verbose-but-correct first cut.

Monomorphize (3,939 lines) eliminates generics by creating specialized copies: `identity<i32>` becomes `identity_i32`. This is iterative — specializing a function may reveal new generic calls in its body. In practice, it converges in two or three iterations. The tradeoff vs. vtables is code size for speed; for a compiler where types are known at compile time, monomorphization wins.

0.57.4 14.4 Emit C

The C backend (`src/c_backend/c_backend.ly`, 5,551 lines — the largest file in the compiler) translates monomorphized LIR into C source.

Type ordering matters. C requires types to be defined before use. The backend topologically sorts struct definitions using Kahn's algorithm, emits forward declarations for all classes, then emits struct definitions, then function definitions. Fieldless enums become C `typedef enum`. Enums with payloads become tagged unions — a `tag` field plus a `union` of variant structs.

Classes become heap-allocated structs. In AoS mode (the default), each class type gets a slab allocator — a block-based free list that avoids per-object `malloc`/`free` overhead. In SoA mode (`--soa`), classes become `uint32_t` handles into parallel arrays, one array per field. The `--soa` flag switches the entire program's class layout without changing a single line of Lyric source.

Lambdas are hoisted to top-level C functions. Captured variables are packed into a context struct that's passed as the first argument. Spawned blocks work the same way — the captured variables become fields of a context struct passed to a `pthread_create` wrapper.

The output is one C file. The compiler's own output — `lyric.c` — is 114,770 lines. GCC compiles it with `-O2` in a few seconds. The resulting binary is the Lyric compiler.

0.57.5 14.5 The Bootstrap

A self-hosting compiler has a peculiar property: it compiles itself. Feed `./lyric` its own 14 source files, and out comes a C file. Compile that C file with GCC, and you have a new `./lyric` binary. Feed *that* binary its own source, and out comes a C file again. If the two C files

are byte-for-byte identical, you've hit a **fixed point** — the compiler has converged on a stable representation of itself.

That fixed point is the definition of self-hosting, and it's the thing that makes the whole edifice non-magical. Each new feature in the language has to keep compiling the compiler, because the compiler is the largest Lyric program in the world and any breakage shows up immediately.

The build commits to this property in two places. First, `lyric.c` — all 114,770 lines of generated C — is checked into the repository. That's how anyone with GCC can build the compiler without already having a working Lyric toolchain:

```
$ make
gcc -std=gnu11 -O2 -w -I runtime -o lyric lyric.c -lm
```

Second, `make update` regenerates `lyric.c` from `src/` using the current binary:

```
BOOTSTRAP_FILES = \
  src/ast/ast.ly src/ast/modules.ly \
  src/lexer/lexer.ly \
  src/parser/parser.ly src/parser/expr_parser.ly \
  src/desugar/desugar.ly \
  src/checker/checker.ly \
  src/lir/lir.ly \
  src/lowerer/lowerer.ly \
  src/optimizer/optimizer.ly \
  src/monomorphizer/monomorphizer.ly \
  src/memory/memory.ly \
  src/c_backend/c_backend.ly \
  src/main/main.ly

update: lyric
    ./lyric compile $(BOOTSTRAP_FILES) -o lyric.c
```

The 14 files are listed explicitly because, as Chapter 13 explained, the compiler uses zero `import` statements — it relies on flat-namespace merging across all files passed on the command line. The Makefile *is* the module graph.

The fixed-point check itself lives in `test_self_compile.sh`:

1. **Stage 0**: build `./lyric` from the checked-in `lyric.c` with GCC.
2. **Stage 1**: that `./lyric` compiles `src/` → `stage2.c`. GCC compiles `stage2.c` → `stage2` binary.
3. **Stage 2**: `stage2` compiles `src/` → `stage3.c`.
4. **Verify**: `diff stage2.c stage3.c` is empty.

When Stage 1 and Stage 2 produce identical C output, the compiler has reached a fixed point — it compiles itself to produce a compiler that compiles itself to produce the same compiler:

```
$ bash test_self_compile.sh
...
  FIXED POINT REACHED - lyric_stage2.c == lyric_stage3.c

=== Verifying checked-in lyric.c is current ===
  lyric.c matches compiler output
```

The reason this matters in practice: when you edit a `.ly` file in `src/`, the *checked-in* `lyric.c` is still the old compiler. You run `make` (rebuilds the binary from old `lyric.c`), `make update` (the old binary compiles your new source into a new `lyric.c`), `make` again (rebuilds the binary from the new `lyric.c`), and now everything matches. If something is subtly wrong — a generated identifier whose hash depends on iteration order, say — the second `lyric.c` won't match what the new binary would produce when run on itself, and `test_self_compile.sh` will print a diff instead of `FIXED POINT REACHED`. That's the regression net.

Origin. The Lyric compiler was not always written in Lyric. The first compiler was written in Go — a few tens of thousands of lines that could parse Lyric, type-check it, lower it, and emit C. The Go compiler compiled the first Lyric-written compiler, GCC compiled that C, and the resulting binary could compile itself. Once the Lyric-written compiler stopped needing the Go version to bootstrap, the Go compiler was retired. The checked-in `lyric.c` replaced it: any future change to the language only needs GCC and the current `lyric.c` to bootstrap. Every PR diffs against `lyric.c`; `.gitattributes` marks it as generated so code review tools collapse it by default.

0.57.6 14.6 The Numbers

The compiler by the numbers:

Component	Lines
`c_backend.ly`	5,551
`checker.ly`	4,919
`monomorphizer.ly`	3,939
`lowerer.ly`	3,669
`memory.ly`	2,855
`main.ly`	1,602
`optimizer.ly`	1,556
`desugar.ly`	1,534
`expr_parser.ly`	1,496
`ast.ly`	1,476
`parser.ly`	1,383
`lir.ly`	943
`modules.ly`	907
`lexer.ly`	679
Compiler total	**32,509**
`stdlib/std.ly`	733
`stdlib/string.ly`	258
Stdlib total	**991**
Grand total	**33,500**

33,500 lines of Lyric (compiler + stdlib) produce 114,770 lines of C. The 3.4× expansion ratio comes from monomorphization (each generic function becomes multiple concrete copies), generated destructors, slab allocator boilerplate, and the verbose nature of C compared to Lyric.

0.57.7 14.6.1 First-Iteration Benchmarks

The Lyric compiler was bootstrapped from a Go compiler. Both implement the same pipeline — parse, check, lower, optimize, monomorphize, emit C — and the Go compiler remains in `legacy/go-compiler/` for comparison. On the day the bootstrap reached its fixed point (June 12, 2026):

Metric	Go compiler	Lyric bootstrap	Delta
Lines of code	33,739	26,813	**−20.5%**
Total bytes	929,693	837,914	**−9.9%**
Bytes per line	27.6	31.2	+13% (longer lines)

The line reduction (20%) exceeds the byte reduction (10%) because Lyric lines are *longer* on average — 31.2 bytes versus 27.6. The savings come from genuine expressiveness, not from cramming more onto each line. Relations eliminate boilerplate that Go requires you to write by hand. Match expressions replace chains of `if`/`else if` with type-checked exhaustive dispatch. The `?` operator replaces Go's three-line `if err != nil { return ..., err }` blocks with a single character.

Compiling this same codebase with the `--soa` flag — switching all class allocation from Array-of-Structs to Struct-of-Arrays layout with zero code changes — delivers an additional **10% speedup and 14% memory reduction** (measured on a MacBook Air M2).

These are first-iteration numbers. The language was fourteen days old. The compiler had not yet been optimized for Lyric idioms — it was a transliteration of the Go original, carrying Go habits into a language that doesn't need them. Every subsequent round of loop engineering on the compiler — rewriting Go patterns into native Lyric — should widen these margins further.

The test suite has 89 `.ly` files under `testdata/`, 83 of them paired with golden output files under `testdata/golden/`. Each test compiles a `.ly` file, runs the resulting binary, and diffs the output against the `.expected` file. The tests cover every feature in this book — enums, match, generics, relations, interfaces, Dict, concurrency, packages, error handling — plus a few that don't (the `--soa` slab layout switch, the `--detect-uaf` debug mode, the `--rc-free` reference-counting mode). The full self-compile fixed-point check in `test_self_compile.sh` is the integration test on top.

0.57.8 14.7 Every Feature Used

The compiler uses every feature this book teaches:

- **Structs** for AST leaf data (field definitions, type parameters, import entries)
- **Enums** for expression kinds, statement kinds, type kinds, token kinds
- **Classes** for AST nodes, checker state, LIR programs, C backend state
- **Match** everywhere — the checker, lowerer, and C backend are largely match expressions over expression and statement kinds
- **Generics** in the standard library types the compiler depends on
- **Relations** for ownership — the AST owns its nodes, the LIR program owns its declarations
- **Interfaces** powering ArrayList, HashedList, and Dict
- **Dict** for symbol tables, type registries, function lookups, class rename maps
- **Sym** for all identifier comparison — the compiler interns every identifier, keyword, and operator
- **Error handling** with `(T, error)` tuples and `?` propagation through the parser
- **Multi-file packages** — 14 `.ly` files across 12 directories, each directory a `lyric <name> { }` block, all flat-merged via `BOOTSTRAP_FILES` on the command line (no `import` statements — see §13.9)
- **F-strings** for error messages and C code generation
- **StringBuilder** for the C backend's output buffer
- **Slices** for parameter lists, field lists, statement blocks, everywhere
- **`mut` parameters** for in-place modification of lowerer and monomorphizer state
- **External methods** for the Dict/ArrayList/Sym method APIs

The compiler is the language's most comprehensive test. If a feature works in the compiler, it works. If it doesn't work in the compiler, it gets fixed — because the compiler can't compile itself until it does.

And that, finally, is what self-hosting buys you. The Lyric you've been learning across these fourteen chapters is the same Lyric that compiles itself, at a fixed point, in a few seconds. Every enum, every relation, every f-string, every `?` propagation — load-bearing. There is no second-class implementation language hiding behind the curtain. You've been reading the source code of the thing that built it.

0.58 Appendix A: Language Quick Reference

0.58.1 Keywords

Declaration keywords:

Keyword	Purpose
`func`	Function declaration
`class`	Heap-allocated reference type
`struct`	Stack-allocated value type
`enum`	Sum type (fieldless or with payloads)
`interface`	Multi-class contract
`relation`	Ownership/reference declaration
`type`	Type alias
`impl`	Interface implementation block
`embed`	Copy fields and destructors from another interface
`import`	Package import
`let`	Variable binding (immutable)
`pub`	Public visibility modifier
`destructor`	Destructor declaration in interfaces

Control flow keywords:

Keyword	Purpose
`if` / `else`	Conditional (expression or statement)
`for` / `in`	Iteration over slices, ranges, generators
`while`	Loop with condition
`match`	Pattern matching (exhaustive)
`return`	Return from function
`break`	Exit loop
`continue`	Skip to next iteration
`case`	Branch in `select`
`select`	Channel multiplexing
`spawn`	Launch concurrent block
`lock`	Scoped mutex acquisition
`yield`	Produce value from generator

Keyword	Purpose
`cascade`	Reserved — currently a no-op statement, slated for removal (use `owns`/`refs` on relations)

Modifier and operator keywords:

Keyword	Purpose
`mut`	Mutable binding or pass-by-reference parameter
`self`	Receiver in method
`as`	Type cast (numeric, unchecked)
`is`	Variant check without destructuring
`where`	Generic constraint clause
`owns`	Cascade-destroy relation
`refs`	Unlink-only relation
`implements`	Declare interface conformance on a class
`permanent`	Class modifier — opts the class out of slab reclamation and ref-counting (Ch 10, Ch 11.5)
`trusted`	Function modifier — allows raw `ref` / `unref` ops inside the body (Ch 8.2, Ch 11.5)
`final`	Function modifier — pre-destruction hook that runs before the auto-generated destructor (Ch 8.7)

Literals:

Keyword	Purpose
`true` / `false`	Boolean literals
`null`	Null literal

Contextual keywords — these are identifiers in most positions, keywords only in specific contexts:

Keyword	Context
`field`	Inside interface blocks: field injection
`lock`	As statement: scoped mutex
`implements`	After class name
`guarded_by`	Annotation on fields

0.58.2 Types

Primitive types:

Type	Size	Description
`i8`, `i16`, `i32`, `i64`	1–8 bytes	Signed integers
`u8`, `u16`, `u32`, `u64`	1–8 bytes	Unsigned integers
`f32`, `f64`	4, 8 bytes	IEEE 754 floating point
`bool`	1 byte	`true` or `false`
`string`	fat pointer	Alias for `[u8]`
`unit`	0 bytes	No value

Composite types:

Syntax	Description
`[T]`	Slice of `T` (data + len + cap)
`T?`	Optional — `T` or `null`
`(T, U)`	Tuple — access via `._0`, `._1`
`T | U`	Union type — matched exhaustively
`func(T) -> U`	Function type
`channel<T>`	Channel for concurrent communication

Type construction:

```
struct Point { x: f64, y: f64 }           // value type, stack-all
class Node { value: i32 }                 // reference type, heap-
enum Color { Red Green Blue }             // fieldless enum (varia
enum Shape {                              // enum with payloads
```

```
    Circle(radius: f64)
    Rect(w: f64, h: f64)
}
type StringList = [string]                    // type alias
```

0.58.3 Operator Precedence

From lowest to highest:

Precedence	Operators	Associativity
1	`||`	Left
2	`&&`	Left
3	`|` (bitwise)	Left
4	`^`	Left
5	`&` (bitwise)	Left
6	`== !=`	Left
7	`< > <= >=`	Left
8	`<< >>`	Left
9	`+ -`	Left
10	`* / %`	Left
11	`-` `!` (unary)	Right (prefix)
12	`. () [] !`	Left (postfix)

Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`

Other operators: `?` (error propagation), `as` (type cast), `is` (variant test)

0.58.4 Built-in Functions

Function	Signature	Description
`println(args...)`	variadic → `unit`	Print with newline
`print(args...)`	variadic → `unit`	Print without newline
`eprintln(args...)`	variadic → `unit`	Print to stderr with newline
`eprint(args...)`	variadic → `unit`	Print to stderr
`len(s)`	`[T]` or `string` → `i32`	Length of slice or string

Function	Signature	Description
`append(s, elem)`	`([T], T) → [T]`	Append element, return new slice
`assert(cond)`	`bool → unit`	Assert with file:line; optional message
`assert_eq(a, b)`	`(T, T) → unit`	Assert equality with file:line
`isnull(x)`	`T? → bool`	Test for null
`panic(msg)`	`string → unit`	Abort with message
`exit(code)`	`i32 → unit`	Exit process
`atoi(s)`	`string → (i64, bool)`	Parse integer from string; `bool` is `false` on parse failure
`itoa(n)`	`i32 → string`	Integer to string
`char_to_string(c)`	`u8 → string`	Single byte to string
`sym(s)`	`string → Sym`	Create interned symbol
`make_channel<T>()`	`() → channel<T>`	Unbuffered channel
`make_channel<T>(n)`	`i32 → channel<T>`	Buffered channel

0.58.5 Built-in Methods

Slice methods (`[T]`):

Method	Return	Description
`.len()`	`i32`	Length
`.push(elem)`	`unit`	Append in place
`.pop()`	`T`	Remove and return last element
`.extend(other)`	`unit`	*Silent no-op today — the lowerer never wired it up. Use `append(xs, elem)` in a loop, or `xs = xs + other` for a new slice.*
`.contains(elem)`	`bool`	Linear search
`.index_of(elem)`	`i32`	First index, or -1
`.sort()`	`unit`	In-place sort
`.remove(idx)`	`unit`	Remove at index
`.first()`	`T?`	First element or null

Method	Return	Description
`.last()`	`T?`	Last element or null
`.is_empty()`	`bool`	True if length is 0
`.join(sep)`	`string`	Join string slices with separator
`.slice(lo, hi)`	`[T]`	Sub-slice

String methods (`string` / `[u8]`):

Method	Return	Description
`.len()`	`i32`	Byte length
`.contains(s)`	`bool`	Substring search
`.has_prefix(s)`	`bool`	Starts with (alias: `.starts_with()`)
`.has_suffix(s)`	`bool`	Ends with (alias: `.ends_with()`)
`.index_of(s)`	`i32`	First occurrence, or -1
`.trim()`	`string`	Strip leading/trailing whitespace
`.split(sep)`	`[string]`	Split on separator
`.replace(old, new)`	`string`	Replace all occurrences
`.repeat(n)`	`string`	Repeat `n` times
`.to_upper()`	`string`	Uppercase
`.to_lower()`	`string`	Lowercase
`.substring(lo, hi)`	`string`	Sub-string by byte index
`.char_at(idx)`	`string`	Single character as string
`.is_empty()`	`bool`	True if length is 0

Channel methods (`channel<T>`):

Method	Return	Description
`.send(val)`	`unit`	Send value (blocks if full)
`.receive()`	`T`	Receive value (blocks if empty)
`.close()`	`unit`	Close channel

0.58.6 String Literals

Syntax	Description
`"hello"`	String literal
`f"x = {expr}"`	F-string with interpolation
`f"use {{braces}}"`	Escaped braces in f-strings
`"""multi-line"""`	Triple-quoted string (no escaping needed)
`'A'`	Character literal → `u8` (value 65)
`` `name` ``	Sym literal (sugar for `sym("name")`)

Escape sequences: `\n` (newline), `\t` (tab), `\\` (backslash), `\"` (quote), `\x41` (hex byte), `\0` (null)

0.58.7 Pattern Matching

```
match expr {
    VariantA(x, y) => { ... }        // destructure enum payload
    VariantB | VariantC => { ... }    // multi-pattern
    val if val > 0 => { ... }         // guard
    Outer(Inner(x)) => { ... }        // nested destructuring
    _ => { ... }                      // wildcard (must be last)
}
```

`match` is an expression — `let x = match ... { ... }`

Conditional extraction:

```
if let x = optional_val { ... }          // bind non-null optional
if let Circle(r) = shape { ... }         // destructure enum varia
let x = optional_val else { return }     // early-exit if null
```

0.58.8 Declarations

```
// Functions
func add(a: i32, b: i32) -> i32 { return a + b }
func T.method(self) -> string { ... }            // external metho

// Lambdas
let f = (x: i32) -> i32 { x * 2 }               // paren-style
let g = |x: i32| -> i32 { x * 2 }               // pipe-style

// Variables
```

```
let x = 42                              // immutable, type inferred
let mut y: f64 = 3.14                   // mutable, type annotated

// Error handling
func parse(s: string) -> (i32, error) { ... }
let val = parse(input)?                 // propagate error, unwrap s

// Generics
func identity<T>(x: T) -> T { return x }
func sort<T: Comparable>(items: [T]) { ... }
func find<K, V>(d: Dict<K, V>, k: K) -> V? where K: Hashable { ...

// Relations
relation ArrayList Team:roster owns [Player:team]
relation HashedList Dict<K,V>:d owns [DictEntry<K,V>:d]

// Interfaces
interface MyList<P, C> {
    field P.children: [C]
    func P.add(self, child: C)
    destructor P {
        for child in self.children {
            child.destroy()
        }
    }
}

// Impl blocks
impl MyList<Team, Player> {
    P.children <-> Team.roster_children
    func P.add(self, child: C) { self.children.push(child) }
}

// Tests
func test_addition() {
    assert_eq(add(2, 3), 5)
}

// Modules
import ast                               // qualified: ast.Node
```

0.58.9 Annotations

The only annotation that the Lyric grammar parses today is `guarded_by(lock_name)` on fields:

```
class Executor {
    active: Dict<u32, Job>   guarded_by(mu)
    mu: lock
}
```

A larger function-level annotation table — `requires:`, `ensures:`, `raises:`, `concurrent:`, `requires_lock`, `excludes_lock`, `spawns:`, `pure:` — is described in the language spec as a roadmap target but does not parse today.

The Context-Driven Development annotations (`why:`, `doc`, `invariant:`, `verified_at:`, `source:`, `fake:`) and the three-zone `.lyric` file layout have moved to the separate **`lyre`** tool — they are not part of the Lyric grammar. See Appendix E.

0.58.10 Toolchain

Command	Purpose
`lyric compile file.ly`	Compile to C, then to binary
`lyric compile <dir>`	Compile entire module rooted at a `lyric.mod` directory
`lyric compile --soa file.ly`	Compile with SoA memory layout
`lyric test file.ly`	Discover and run `test_*` functions
`lyric fmt file.lyric`	Format `.lyric` design files
`lyric help`	Show usage

The CDD-layer commands `lyric verify`, `lyric update`, and `lyric gen` live in the separate `lyre` tool — see Appendix E.

0.59 Appendix B: Standard Library Reference

The standard library is two files: `stdlib/std.ly` (733 lines) and `stdlib/string.ly` (258 lines) — 991 lines total. Both are auto-imported into every program — no `import` needed. Everything here

is written in Lyric itself, using the same interfaces and relations covered in Chapters 8 and 9.

0.59.1 Relation Interfaces

These are multi-class interfaces (Chapter 9) that define ownership patterns. You name one on a `relation` line and the desugar pass injects fields, methods, and destructors onto the participating classes — you never write an `impl` block for them yourself. The three user-facing hints are `ArrayList`, `DoublyLinked`, and `HashedList`. (Any user-defined binary interface can also serve as a hint; see spec §Relations.)

0.59.1.1 ArrayList<P, C> Dynamic array of children with O(1) swap-remove. The most common relation type, and the default when you don't know which list shape you want.

Injected fields:

Field	Type	Description
`P.children`	`[C]`	The parent's array of children
`C.parent`	`P?`	Back-reference to owning parent
`C.index`	`i32`	Position in parent's array

The injected field names are prefixed with the relation label; for `relation ArrayList Team:roster owns [Player:team]` you get `t.roster_children`, `p.team_parent`, `p.team_index`.

Generated methods (label-prefixed using the parent relation label, e.g. `:roster` below):

Method	Description
`parent.roster_append(child)`	Append child to parent's array
`parent.roster_remove(child)`	Swap-remove child from parent's array (O(1))

The desugared free-function form is `array_append<Team, Player>(t, p)` / `array_remove<Team, Player>(p)`. The label-prefixed method form is the idiomatic call site.

Destructor: When the parent is destroyed, all children in the array are cascade-destroyed (for `owns`) or unlinked (for `refs`); the `owns` vs `refs` choice on the `relation` line selects the variant.

Example:

```
class Team { name: string }
class Player { name: string }
relation ArrayList Team:roster owns [Player:team]

func main() {
    let t = Team { name: "Sharks" }
    let p1 = Player { name: "Alice" }
    let p2 = Player { name: "Bob" }
    t.roster_append(p1)
    t.roster_append(p2)
    println(f"{t.roster_children.len()}")  // 2
    t.roster_remove(p1)                    // O(1) swap-remove
    println(f"{t.roster_children.len()}")  // 1
    t.destroy()                            // destroys remaining p
}
```

`roster_remove` uses swap-remove: the last element takes the removed element's slot, and the array shrinks by one. Order is not preserved, but removal is O(1).

0.59.1.2 DoublyLinked<P, C> Intrusive doubly-linked list. Insertion preserves order; removal is O(1) without swap (siblings get relinked). Use when iteration order needs to be stable across removals, when you want to thread one child through several parents at once (each list lives on a separate label), or when you'd otherwise pay for swap-remove churn.

Injected fields:

Field	Type	Description
`P.first`	`C?`	Head of the list
`P.last`	`C?`	Tail of the list
`C.parent`	`P?`	Back-reference to owner
`C.next`	`C?`	Next sibling
`C.prev`	`C?`	Previous sibling

Generated methods (label-prefixed):

Method	Description
`parent.label_append(child)`	Append child to end of list
`parent.label_remove(child)`	Unlink child from list (O(1))

The desugared free-function form is `dll_append<P, C>(parent, child)` / `dll_remove<P, C>(child)`. As with `ArrayList`, the `owns` vs `refs` choice on the `relation` line selects whether parent destruction cascades.

Example (excerpted from `testdata/graph.ly`):

```
class Network { name: string }
class Person   { handle: string }
class Follow   { since: i64; weight: f64 = 1.0 }

relation DoublyLinked Network:nodes owns [Person:graph]
relation DoublyLinked Person:out    refs [Follow:source]
relation DoublyLinked Person:in     refs [Follow:target]
```

Each `Person` is in three different lists (the network's `nodes` list, plus their outgoing and incoming `Follow` lists) — the relation labels keep the injected fields and methods (`nodes_first`, `out_append`, `in_remove`, …) distinct.

0.59.1.3 HashedList<P, C> Hash table with linear probing and 75% load factor. The backbone of `Dict` and `SymTable`.

Injected fields:

Field	Type	Description
`P.children`	`[C]`	Dense array of entries
`P.buckets`	`[i32]`	Bucket-to-index map (-1 = empty, -2 = tombstone)
`P.hash_cap`	`i32`	Current bucket array capacity
`P.hash_count`	`i32`	Number of live entries
`C.parent`	`P?`	Back-reference to owner
`C.index`	`i32`	Position in children array

Children must implement a `hash_key(self) -> u64` method. The interface uses this for bucket placement.

Functions:

Function	Description
`hash_insert(parent, child)`	Insert or replace by hash key (auto-init, auto-rehash)
`hash_lookup(parent, key) -> C?`	Find entry by hash value
`hash_remove(parent, key) -> bool`	Remove entry by hash value
`hash_init(parent, cap)`	Initialize with given capacity (rarely needed; auto on first insert)

The table grows (doubles capacity) when load exceeds 75%. Tombstones (-2) are used for deletion to preserve linear probe chains.

> *Match semantics:* `HashedList` matches an entry by `hash_key()` alone — `Hashable` currently has only `get_hash()` and no `equals()`. For `Sym` keys this is collision-safe by construction (the intern table enforces one entry per string). For other key types, two distinct keys that hash to the same `u64` will silently match the wrong entry. Restoring `Hashable.equals` is on the roadmap.

0.59.2 Sym

Interned string symbol. Hash computed once at creation; comparisons are pointer equality (O(1)).

```
let s1 = sym("hello")
let s2 = `hello`          // backtick syntax - same as sym("hello"
assert(s1 == s2, "same")  // pointer equality - same interned inst
```

Methods:

Method	Return	Description
`.get_name()`	`string`	The original string

Method	Return	Description
`.get_hash()`	`u64`	Precomputed hash (implements `Hashable`)
`.hash_key()`	`u64`	Same as `get_hash` — used by `HashedList`
`.equals(other)`	`bool`	Pointer equality check (external method on `Sym`)

All `Sym` instances are owned by a global `SymTable` via `HashedList`. The table is declared `pub permanent class SymTable { }` — `permanent` opts the class out of slab reclamation so interned symbols live for the whole program. Calling `sym("x")` twice returns the same instance.

Why Sym exists: `string` deliberately does not implement `Hashable`. If you want to use a string as a hash key, you must wrap it with `sym()`. This forces the hash-once discipline — you never accidentally hash the same string twice in a hot loop.

0.59.3 Hashable

```
interface Hashable {
    func get_hash(self) -> u64
}
```

The constraint required for `Dict` keys. `Sym` implements it; so do all integer types (`i8`–`i64`, `u8`–`u64`). `string` does not — by design.

> *Roadmap:* `Hashable.equals(self, other) -> bool` is planned but not present today. Until it lands, `HashedList` matches entries by hash alone — collision-safe for `Sym` keys (the intern table guarantees uniqueness) but not for arbitrary `Hashable` types.

0.59.4 Dict<K, V>

Generic hash table. Keys must satisfy the `Hashable` constraint.

```
let d = Dict<Sym, i32>()
d.set(`x`, 10)
d.set(`y`, 20)
```

```
if d.has(`x`) {
    let entry = d.get(`x`)!
    println(f"{entry.value}")    // 10
}

d.remove(`x`)
println(f"{d.keys().len()}")    // 1
```

Constructor: `Dict<K, V>()` — creates an empty dictionary.

Methods:

Method	Return	Description
`.set(key, value)`	`unit`	Insert or update entry
`.get(key)`	`DictEntry<K,V>?`	Look up entry — null if not found
`.has(key)`	`bool`	Check if key exists
`.remove(key)`	`bool`	Remove entry, returns true if found
`.keys()`	`[K]`	All keys as a slice
`.length()`	`i32`	Number of entries

`.get()` returns a `DictEntry<K,V>?`, not the value directly. Access the value via `.value`:

```
let entry = d.get(`x`)
if !isnull(entry) {
    println(f"{entry!.value}")
}
```

> *Roadmap:* The stdlib spells the size method `d.length()`, while the spec's Dict table lists it as `len()`; the two will reconcile. Until they do, `d.keys().len()` is portable — `[K]` slices always have `.len()`.

Dict literal syntax. `Dict<K, V>` can be constructed with a brace literal whose first key is a backtick sym or an integer literal:

```
let names = {`alice`: 1, `bob`: 2}        // Dict<Sym, i32>
let lookup = {1: "one", 2: "two"}          // Dict<i32, string>
let empty: Dict<Sym, string> = {}          // type annotation need
```

Roadmap: String-literal-keyed Dict literals (`{"NYC": 8_000_000}`) are listed in the spec but the parser currently commits to a struct-literal interpretation when it sees a string literal after `{` and rejects the closing brace. Workaround: `let d = Dict<string, V>()` plus explicit `.set(...)`.

Implementation: Dict is built entirely in Lyric using HashedList:

```
class Dict<K, V> where K: Hashable { }
relation HashedList Dict<K, V>:d owns [DictEntry<K, V>:d]
```

`DictEntry<K,V>` holds `key: K` and `value: V`. The `.set()` method creates a `DictEntry`, and `hash_insert` places it in the table.

0.59.5 StringBuilder

Efficient string builder using `append()` with doubling growth — avoids the O(n^2) cost of repeated string concatenation.

```
let sb = new_string_builder()
sb.write("hello")
sb.write_byte(' ')
sb.write("world")
println(sb.to_string())   // hello world
println(f"{sb.len()}")    // 11
```

Constructor: `new_string_builder()` — returns an empty `StringBuilder`.

Methods:

Method	Return	Description
`.write(s)`	`unit`	Append a string
`.write_byte(b)`	`unit`	Append a single byte
`.to_string()`	`string`	Get the built string
`.len()`	`i32`	Current byte length

0.59.6 Error

The stdlib provides a concrete `Error` class and the `error` interface. Any class with a `message(self) -> string` method satisfies `error` via structural subtyping.

The idiomatic way to construct an error value is the class literal `Error { msg: "..." }`:

```
func divide(a: i32, b: i32) -> (i32, error) {
    if b == 0 {
        return 0, Error { msg: "division by zero" }
    }
    return a / b, null
}
```

Stringify an `error`-typed value with an f-string: `f"{err}"`. (See Ch 5 for why `err.message()` doesn't compile today on `error`-typed values, and why custom `message()` overrides are bypassed by f-string interpolation until the `error` interface gets real dynamic dispatch.)

For a custom error type, define a class with `message(self) -> string` and use it as the concrete return type when you want callers to dispatch to fields directly:

```
class ParseError {
    msg: string
    line: i32

    pub func message(self) -> string {
        return f"line {self.line}: {self.msg}"
    }
}
```

> *Roadmap:* The spec lists `new_error(msg: string) -> error` as a one-liner shortcut, but the C backend doesn't lower it today — calls to `new_error(...)` reach `gcc` undeclared and fail to link. Use the `Error { msg: "..." }` class literal until the lowering lands.

0.59.7 String Utilities

All functions in `stdlib/string.ly`. Since `string = [u8]`, these operate on byte slices.

Search:

Function	Signature	Description
`str_contains`	`(haystack, needle) -> bool`	Substring search
`str_index_of`	`(haystack, needle) -> i32`	First index, or -1
`str_has_prefix`	`(s, prefix) -> bool`	Starts with
`str_has_suffix`	`(s, suffix) -> bool`	Ends with

Transformation:

Function	Signature	Description
`str_replace`	`(s, old, new) -> string`	Replace all occurrences
`str_to_upper`	`(s) -> string`	Uppercase ASCII
`str_to_lower`	`(s) -> string`	Lowercase ASCII
`str_trim`	`(s) -> string`	Strip leading and trailing whi
`str_trim_left`	`(s) -> string`	Strip leading whitespace
`str_trim_right`	`(s) -> string`	Strip trailing whitespace

Splitting and joining:

Function	Signature	Description
`str_split`	`(s, sep) -> [string]`	Split on separator
`str_split_n`	`(s, sep, n) -> [string]`	Split into at most `n` parts
`str_join`	`(parts, sep) -> string`	Join with separator
`str_concat`	`(a, b) -> string`	Concatenate two strings
`str_repeat`	`(s, n) -> string`	Repeat `n` times

Conversion:

Function	Signature	Description
`itoa`	`(n: i64) -> string`	Integer to string

Function	Signature	Description
`atoi`	`(s: string) -> (i64, bool)`	String to integer; `bool` is `false` on parse failure
`parse_float`	`(s: string) -> (f64, bool)`	String to float; `bool` is `false` on parse failure
`char_to_string`	`(c: u8) -> string`	Single byte to string
`parse_int`	`(s: string) -> i64`	Stdlib lenient int parser — returns 0 on failure
`str_to_float`	`(s: string) -> f64`	Stdlib lenient float parser — returns 0.0 on failure

Utility:

Function	Signature	Description
`hash_string`	`(s: string) -> u64`	FNV-1a hash of a string
`push_bytes`	`(mut dst, src)`	Bulk append bytes in place

0.59.8 Other Globals

Function	Signature	Description
`range`	`(start, end) -> gen i32`	Generator yielding integers from `start` to `end - 1`

0.59.9 I/O Built-ins

I/O is currently a small set of built-in functions wired directly into the lowerer and C backend, not a stdlib library. The set is the minimum needed to bootstrap the compiler: read and write whole files, run external commands, manipulate paths.

Files and processes:

Function	Signature	Description
`read_file`	`(path: string) -> (string, bool)`	Read whole file; `bool` is `true` on success
`write_file`	`(path: string, content: string) -> bool`	Write whole file; `true` on success
`file_exists`	`(path: string) -> bool`	Check whether a path exists
`list_dir`	`(path: string) -> ([string], bool)`	List directory entries
`mkdtemp`	`(prefix?: string) -> string`	Create a temporary directory
`exec_command`	`(name: string, args: [string]) -> (string, bool)`	Run an external command, capture stdout

Path manipulation:

Function	Signature	Description
`path_join`	`(parts: [string]) -> string`	Join path components
`path_dir`	`(p: string) -> string`	Directory portion of a path
`path_base`	`(p: string) -> string`	Base filename portion
`path_ext`	`(p: string) -> string`	File extension

Process and environment:

Function	Signature	Description
`os_args`	`() -> [string]`	Command-line arguments
`os_getwd`	`() -> string`	Current working directory
`os_exit`	`(code: i32) -> unit`	Exit the process
`exit`	`(code: i32) -> unit`	Alias of `os_exit`
`panic`	`(msg: string) -> unit`	Print message and abort

Output:

Function	Signature	Description
`println` / `print`	`(...) -> unit`	Print to stdout (`println` adds newline)
`eprintln` / `eprint`	`(...) -> unit`	Print to stderr

> *Roadmap:* A proper I/O library with `Reader` / `Writer` interfaces, a `File` class with streaming reads and writes, `BufferedReader` / `BufferedWriter`, directory operations with structured errors, and stdin/stdout/stderr as `Reader`/`Writer` values is sketched in the spec (§I/O Library — Planned). For now, "open a file, read it all, do something with the string" is the supported shape.

0.60 Appendix C: The Lyric Toolchain

The Lyric toolchain is one binary: `lyric`. It compiles, tests, and formats. There's no build system, no package manager, no linker invocation. This appendix documents every command.

0.60.1 C.1 lyric compile

```
lyric compile <file.ly | dir> [...] [-o output.c] [--soa] [--detec
```

Compile one or more `.ly` files (or a single module directory) to C:

```
$ lyric compile calculator.ly -o calculator.c
```

The output is a single `.c` file containing your entire program, the standard library, and a `main()` entry point. To produce a binary:

```
$ gcc -std=gnu11 -O2 -w -o calculator calculator.c -lm -lpthread
$ ./calculator
```

The `-std=gnu11` is required — the generated C uses GNU statement expressions for some lowering patterns. `-lm` provides math functions, `-lpthread` provides threading primitives for `spawn` and channels.

Flags:

`-o output.c` — Set the output filename. Without it, the compiler derives a name from the first input file — `calculator.ly` produces `calculator.c`.

`--soa` — Switch all class allocation from Array-of-Structs to Struct-of-Arrays layout. Your code doesn't change. Chapter 11 explains the performance implications: 10% faster, 14% less memory on data-intensive workloads.

`--detect-uaf` — Use-after-free debug mode. Freed class slots are poisoned with a sentinel refcount and every subsequent access checks for it. Slower, but turns "silent UAF" into a clear runtime abort. See Chapter 11 §11.4.

`--rc-free` (default ON) — When a refcounted class's refcount reaches zero, run its destructor and release the slot. This is the normal refcounting regime described in Chapter 11 §11.5.

`--no-rc` — Disable the RC=0-triggers-destroy step. Refcounts are still maintained, but slots are not reclaimed until the owning relation cascades or the program exits. Useful for isolating refcount bugs from destructor bugs.

`--lir-dump file` — Dump the LIR (Low-level Intermediate Representation) to the named file before C emission. Useful for debugging the compiler itself.

`--c` is accepted as a legacy no-op for backward compatibility.

Module mode:

```
$ lyric compile . -o out.c
```

If the argument is a directory containing a `lyric.mod` file, the compiler switches to module mode: it walks the top-level `.ly` files in that directory and resolves the root file's `import` statements. Module mode is single-level today — `import` statements in imported packages are not recursively resolved (see Chapter 13 §13.7).

For multi-directory projects whose imports outgrow single-level resolution, the working pattern is the **flat file list** — pass every `.ly` file on one command line, all merged into a single namespace. The Lyric compiler itself is built this way: `make update` runs `./lyric compile $(BOOTSTRAP_FILES) -o lyric.c`, where `BOOTSTRAP_FILES` enumerates all 14 source files across 12 directories

with zero `import` statements between them. Chapters 13 §13.9 and 14 §14.5 cover the bootstrap in detail.

0.60.2 C.2 lyric test

```
lyric test <file.ly> [...] [-o output.c] [--soa] [--detect-uaf] [-
```

Compile, discover test functions, and run them:

```
$ lyric test test_lexer.ly
PASS  test_tokenize_number
PASS  test_tokenize_operators
PASS  test_tokenize_parens
3 tests, 3 passed, 0 failed
```

The test command compiles your files to C, links them with GCC (using `-O0 -g` for debuggability), then runs the resulting binary. It discovers test functions by scanning for any function whose name starts with `test_` — no framework, no registration, no annotations.

The generated runner calls each `test_*` function in source order and tracks pass/fail counts. The runtime `assert` and `assert_eq` macros `longjmp` back to a `setjmp` in the runner on failure, so a failed assertion ends *that* test but the suite continues with the next one. (Outside the test runner — e.g. an `assert` in `main()` of a regular compile — the macros fall back to `exit(1)`, since there's no jump buffer to land in.) A segfault in one test still aborts the whole binary. The exit code is non-zero if any test failed.

Roadmap: parallel execution, test filtering, and per-test timing are all planned. None of them are implemented today.

The same backend flags accepted by `compile` (`--soa`, `--detect-uaf`, `--rc-free` / `--no-rc`, `--lir-dump`, `-o`) are accepted here too — useful when you want to run a test suite under SoA or UAF-detection mode without changing your build script. The `-o` flag writes the generated C to disk and exits without running, for when you want to inspect what the runner looks like.

The test command accepts the same file arguments as `compile`. You can pass multiple files, and all `test_*` functions across all files will be discovered and run.

0.60.3 C.3 lyric fmt

```
lyric fmt <file.lyric> [...]
```

Format `.lyric` design files (not `.ly` source files):

```
$ lyric fmt ast.lyric checker.lyric
formatted ast.lyric
formatted checker.lyric
```

The formatter normalizes whitespace, sorts declarations by their original source order, and preserves comments. It's idempotent — running it twice produces the same output.

Note that `fmt` operates on `.lyric` design files, not `.ly` source files. There is no source formatter yet. The `.lyric` files are the declaration-only Lyric artifacts described in Chapter 13 §13.8 — the same files consumed by the `lyre` toolchain documented in Appendix E.

0.60.4 C.4 lyric help

```
$ lyric help
Usage: lyric <command> [arguments]

Commands:
  compile  <file.ly> [...] [-o out]   Compile .ly files to C
  test     <file.ly> [...]             Compile, discover test_* fu
  fmt      <file.lyric> [...]          Format .lyric files
  help                                 Show this message
```

Also available as `lyric -h` or `lyric --help`.

Command prefix matching: The CLI accepts unique prefixes — `lyric c` resolves to `compile`, `lyric t` to `test`, `lyric f` to `fmt`. If a prefix is ambiguous, the compiler reports the matching commands and exits.

0.60.5 C.5 The Generated C

The C output is self-contained. It includes:

- A runtime header (`lyric_runtime.h`) with string operations, slab allocator macros, channel primitives, and the test-runner assertion macros

- All type definitions: forward declarations, then structs in topological order, then tagged unions for enums
- All function bodies, monomorphized — generic functions are expanded into concrete copies per instantiation
- Slab allocator globals for each class (one free-list and block array per type)
- A `main()` that calls your `main()` (or, in test mode, a generated runner that invokes each `test_*` function in source order)

The output compiles cleanly with GCC and Clang. The `-w` flag suppresses warnings — the generated C is correct but not pretty, and compilers occasionally warn about unused variables from monomorphization.

Compilation performance: The Lyric compiler itself — 33,500 lines of Lyric (the compiler in `src/` plus the auto-imported `stdlib/`) across 14 files in 12 directories — compiles to 114,770 lines of C in about 0.2 seconds on a modern laptop. GCC compiles that C file in a few seconds. The total from-source-to-binary time is under 5 seconds.

0.60.6 C.6 The Bootstrap

There is no pre-built `lyric` binary to download. The compiler bootstraps from the checked-in `lyric.c`:

```
$ make
gcc -std=gnu11 -O2 -w -I runtime -o lyric lyric.c -lm
```

`lyric.c` is the canonical compiler output, produced by the Lyric compiler compiling itself. To regenerate it from `src/` after editing the compiler:

```
$ make update
./lyric compile $(BOOTSTRAP_FILES) -o lyric.c
lyric.c updated (114770 lines)
```

`BOOTSTRAP_FILES` is the explicit list of 14 `.ly` files that make up the compiler, passed as a flat file list (not module mode — see Chapter 13 §13.9).

To verify the fixed-point property — that the compiler faithfully compiles its own source:

```
$ make self-test
$ # or, equivalently:
$ bash test_self_compile.sh
```

`test_self_compile.sh` builds Stage 0 from the checked-in `lyric.c`, has that binary self-compile to `stage2.c`, then builds a Stage 2 binary that compiles `src/` again to `stage3.c`, and finally `diff stage2.c stage3.c` — which must be empty. If the diff is empty, the compiler is at a fixed point. This is verified in CI on every change. See Chapter 14 §14.5 for the full bootstrap story.

0.61 Appendix D: From Go/Rust/C++ to Lyric

This appendix maps concepts you already know to their Lyric equivalents. It's a translation guide, not a tutorial — every feature listed here is explained fully in the chapters referenced.

0.61.1 D.1 Type Declarations

Concept	Go	Rust	C++	Lyric
Value type with fields	`type Point struct { X, Y int }`	`struct Point { x: i32, y: i32 }`	`struct Point { int x, y; };`	`struct Point { x: i32 y: i32 }`
Heap-allocated type	— (use `new`)	`struct Point { ... } + Box<Point>`	`class Point { ... };`	`class Point { x: i32 y: i32 }`
Enum (fieldless)	`const ( Red = iota; Blue )`	`enum Color { Red, Blue }`	`enum Color { Red, Blue };`	`enum Color { Red Blue }`
Enum (with data)	— (use interfaces)	`enum Shape { Circle(f64) }`	`std::variant<Circle, Rect>`	`enum Shape { Circle(r: f64) }`
Type alias	`type Name = string`	`type Name = String;`	`using Name = std::string;`	`type Name = string`

Key differences: Lyric structs are always value types — copied on assignment, allocated on the stack. Lyric classes are always heap-allocated with identity. There's no choice to make: if you need identity and methods with `self`, use a class. If you need a plain data bag, use a struct. (Chapters 2-3)

0.61.2 D.2 Variables and Mutability

Concept	Go	Rust	C++	Lyric
Immutable binding	—	`let x = 5;`	`const int x = 5;`	`let x = 5`
Mutable binding	`x := 5`	`let mut x = 5;`	`int x = 5;`	`let mut x = 5`
Pass struct by mutable ref	pointer `f(&p)`	`f(&mut p)`	`f(p)` (reference)	`f(mut p)` — `mut` at both call and declaration site
Null	`null`	`None`	`nullptr`	`null`
Optional type	pointer `*T`	`Option<T>`	`std::optional<T>`	`T?`
Unwrap optional	`*p` (no safety)	`.unwrap()`	`.value()`	`x!`

Key difference: `mut` on function parameters means pass-by-mutable-reference. Both the caller and callee must agree — `func translate(mut p: Point, dx: i32)` is called as `translate(mut pt, 5)`. This applies to structs only; classes are already references. (Chapter 3)

0.61.3 D.3 Error Handling

Concept	Go	Rust	C++	Lyric
Error return	`(T, error)`	`Result<T, E>`	exceptions / `std::expected`	`(T, error)`

Concept	Go	Rust	C++	Lyric
Propagate error	`if err != nil { return ..., err }`	`?`	`throw` / manual	`?`
Error type	`error` interface	`Error` trait	`std::exception`	`error` interface — any class with `message(self) -> string`
Quick error	`fmt.Errorf("...")`	`Err("...")`	`throw std::runtime_error("...")`	`Error { msg: "..." }`

Lyric's error model is Go's tuples plus Rust's `?` operator. You get explicit error types in signatures (no hidden exceptions) with concise propagation (no three-line `if err != nil` blocks). (Chapter 5)

0.61.4 D.4 Generics

Concept	Go	Rust	C++	Lyric
Generic function	`func F[T any](x T) T`	`fn f<T>(x: T) -> T`	`template<typename T> T f(T x)`	`f<T>(x: T) -> T`
Constraint	`[T comparable]`	`T: PartialOrd`	`requires std::totally_ordered<T>`	`<T: Comparable>`
Where clause	—	`where T: Hash`	`requires` clause	`where T: Hashable`
Implementation	type params at runtime	monomorphization	template instantiation	monomorphization

Key difference: Lyric requires explicit `<T>` on declarations — you

can't accidentally introduce a type variable by misspelling a type name. At call sites, inference works normally: `identity(42)` infers `T = i32`. (Chapter 6)

0.61.5 D.5 Interfaces and Polymorphism

Concept	Go	Rust	C++	Lyric
Interface/trait	`type Writer interface { Write([]byte) }`	`trait Write { fn write(&self, buf: &[u8]); }`	`class Writer { virtual void write() = 0; };`	`interface Writable { func write(self, data: [u8]) }`
Satisfaction	structural (implicit)	`impl Write for File`	inheritance	structural + optional `implements` annotation
Multi-type interface	—	— (workaround: associated types)	—	`interface Graph<G, N, E> { func G.nodes(self) -> [N] }`
Default methods	—	`fn default() { ... }`	virtual with body	`func count(parent: P) -> i32 { ... }` in interface body
Field injection	—	—	—	`field P.children: [C]` in interface body

Concept	Go	Rust	C++	Lyric
Method on type defined outside class body	`func (c *Counter) Reset()` (any file in same package)	`impl Counter { fn reset(&self) { ... } }`	`void Counter::reset() { ... }` (declared in header)	`func Counter.reset(self) { ... }` (external method — Ch 3.5)

This is the big one. No other language has multi-class interfaces. In Go, Rust, and C++, an interface describes one type. Lyric interfaces can span multiple type parameters — `Graph<G, N, E>` defines methods on `G`, `N`, and `E` simultaneously. One `impl` block binds all three to concrete types. Monomorphization means zero runtime cost. (Chapters 8-9)

0.61.6 D.6 Ownership and Memory

Concept	Go	Rust	C++	Lyric
Memory safety	GC	borrow checker	manual / smart pointers	relations
Ownership declaration	—	single owner by default	`unique_ptr<T>`	`relation ArrayList Parent:label owns [Child:label]`
Non-owning reference	—	`&T` / `Rc<T>`	raw pointer / `shared_ptr`	`relation RefList Parent:label refs [Child:label]`

Concept	Go	Rust	C++	Lyric
Destructor	— (finalizers, unreliable)	`impl Drop`	`~ClassName()`	auto-generated from `owns` relations; `final func cleanup(self)` for user code at destruction
Cascade delete	—	manual	manual	automatic — destroying parent destroys all `owns` children
Scope-exit cleanup	`defer f.Close()`	RAII / `impl Drop`	RAII / destructors	scope-exit destructors (automatic) + `final func` for explicit cleanup; **no `defer` keyword**

The pitch: In C++ you write destructors and get them wrong. In Rust you fight the borrow checker. In Go you accept GC pauses. In Lyric you declare `relation ArrayList Team:roster owns [Player:team]` — one line — and the compiler generates all destructors, parent/child fields, add/remove functions, and cascade delete. Thirty years of proof in production EDA tools. (Chapter 8)

0.61.7 D.7 Strings and Collections

Concept	Go	Rust	C++	Lyric
String type	`string` (immutable)	`String` / `&str`	`std::string`	`string` = `[u8]`
String indexing	`s[i]` → byte	`s.as_bytes()[i]`	`s[i]` → char	`s[i]` → `u8`
String interpolation	`fmt.Sprintf`	`format!()`	— (no built-in)	`f"value is {x}"`
Dynamic array	`[]T` (slice)	`Vec<T>`	`std::vector<T>`	`[T]` (slice)
Hash map	`map[K]V`	`HashMap<K,V>`	`std::unordered_map<K,V>`	`Dict<K,V>` (stdlib, K must be `Hashable`)
Hash key	any comparable	`Hash + Eq`	`std::hash<K>`	`Sym` (interned string) or custom `Hashable` impl

Key difference: `string` is an alias for `[u8]`. There's no separate string type. `Dict` requires keys to implement `Hashable`, and `string` deliberately does not — use `sym("key")` or backtick `` `key` `` syntax to force hash-once discipline. (Chapters 4, 10)

0.61.8 D.8 Concurrency

Concept	Go	Rust	C++	Lyric
Spawn concurrent work	`go func() { ... }()`	`tokio::spawn(async { ... })`	`std::thread t(f)`	`spawn { ... }`
Channel	`ch := make(chan T)`	`mpsc::channel()`	—	`let ch = make_channel<T>()`

Concept	Go	Rust	C++	Lyric
Send/receive	`ch <- v` / `v = <-ch`	`tx.send(v)` / `rx.recv()`	—	`ch.send(v)` / `ch.receive()`
Select	`select { case ... }`	`tokio::select!`		`select { case v = ch.receive() => ... }`
Mutex	`sync.Mutex`	`std::sync::Mutex<T>`		`lock` type + `lock(mu) { ... }`

Lyric's concurrency is Go's model with method syntax for channels. `spawn` captures variables from the enclosing scope automatically — but *by pointer*, which means a captured `let mut counter: i32 = 0` mutated from two `spawn` blocks is a textbook data race. Channels are class pointers with internal locking and are the recommended primitive for cross-spawn communication; for shared mutable values, use `lock` from §12.5 or funnel writes through a channel. (*Roadmap: copy-by-value captures with explicit shared-mutation through channels or locks — see Chapter 12 §12.1.*) (Chapter 12)

0.61.9 D.9 Modules

Concept	Go	Rust	C++	Lyric
Module file	`go.mod`	`Cargo.toml`	`CMakeLists.txt`	`lyric.mod`
Package unit	directory	file (with `mod`)	file / target	directory
Import	`import "pkg"`	`use crate::pkg`	`#include`	`import pkg from "pkg"`
Visibility	uppercase = exported	`pub`	`public:`	`pub`
Build	`go build`	`cargo build`	`cmake --build`	`lyric compile .`

Lyric compiles everything into a single C file — no separate compilation, no linking step, no build system. Module boundaries exist for

namespace management, not compilation units. (Chapter 13)

0.61.10 D.10 What's New — No Equivalent in Other Languages

These features have no direct translation from Go, Rust, or C++:

Relations — Declare ownership graphs; compiler generates all lifecycle code. (Chapter 8)

```
relation ArrayList AST:children owns [Node:parent]
```

Multi-class interfaces — One interface spanning multiple types. (Chapter 9)

```
interface Graph<G, N, E> {
    func G.nodes(self) -> [N]
    func N.edges(self) -> [E]
    func E.target(self) -> N
}
```

`--soa` flag — Switch all class allocation to Struct-of-Arrays layout with no code changes. 10% faster, 14% less memory. (Chapter 11)

`.lyric` sibling artifacts — Declaration-only Lyric files (no function bodies) consumed by the **lyre** toolchain, which layers Context-Driven Development annotations (`why:`, `doc`, `invariant:`, `source:`, `fake:`) on top. These annotations are **lyre features, not Lyric features** — they never appear in `.ly` source. See Chapter 13 §13.8 for the language-side framing and Appendix E for the full lyre walkthrough.

`embed` — Copy fields and destructors from one interface into another. Not inheritance — flat composition at compile time. Methods stay abstract bindings; use `where Iface<...>` to pull in default-method behavior. (Chapter 9)

0.62 Appendix E: The CDD Layer (`lyre`)

Earlier drafts of this book described **Context-Driven Development** — the practice of keeping a `.lyric` design artifact alongside every source file, annotated with `why:` purpose statements, `doc "..."` narrative blocks, `invariant:` claims, and `source:`/`fake:` links to implementation — as if it were part of the Lyric language. It isn't,

and it never was: those annotations don't parse with the Lyric grammar. They are a layer on top, owned by a separate tool called **lyre**.

The split is clean:

- **Lyric** (this book) is the language and compiler. A `.lyric` file, from Lyric's perspective, is a valid Lyric source file with no function bodies — declarations only.
- **lyre** is the design-artifact tool. It reads `.lyric` files, recognizes the CDD annotation layer, and keeps each artifact in sync with the implementation it describes.

lyre's reach is broader than Lyric. The same `.lyric` format that describes a Lyric module can describe a Go package, a Python package, or a TypeScript module — lyre ships extractors for all four ecosystems (`pkg/extract/golang`, `pkg/extract/python`, `pkg/extract/typescript`, `pkg/extract/lyric`). The CDD methodology stands on its own and applies whether the implementation is Lyric, Go, Python, or TypeScript. Lyric is one of lyre's targets, not its only one.

lyre's command surface, for orientation:

Command	Purpose
`lyre gen <package-dir>`	Scaffold a fresh `.lyric` file by extracting declarations from source.
`lyre update <file.lyric>`	Re-extract the auto-generated zones (function index, dependency table) without touching hand-written CDD prose.
`lyre verify <file.lyric>`	Check that the declarations and CDD claims still match the implementation.
`lyre lint <file.lyric>`	Report recoverable quality issues (missing `why:`, broken `source:` links, stale `verified_at:`).
`lyre fmt <file.lyric>`	Format `.lyric` files using Lyric grammar rules.

That's the entire interface. There's no `lyre compile`, no `lyre run` — lyre doesn't produce executable code, it produces and verifies design artifacts. The build still goes through `lyric compile`, `go build`, `pytest`, `tsc`, or whatever the implementation language already uses.

What stays in Lyric proper:

- The `.lyric` file *format* (declaration-only Lyric source) is a Lyric concept — lyre piggybacks on the Lyric grammar to keep `.lyric` files trivially parseable for the Lyric-implementation case.
- The one annotation Lyric's own grammar parses today is `guarded_by(lock_name)` on fields (Chapter 12).
- A roadmap table of function-level annotations (`requires:`, `ensures:`, `concurrent:`, `requires_lock`, `excludes_lock`, etc.) is described in the language spec but does not parse today.

For the full CDD methodology — the three-zone file layout, the `why:/doc/invariant:/verified_at:/source:/fake:` vocabulary, the extractor/verifier workflow per supported language — see the lyre repository and its own documentation. The Lyric toolchain itself ships exactly the four subcommands documented in Appendix C (`compile`, `test`, `fmt`, `help`); everything in this appendix belongs to lyre.