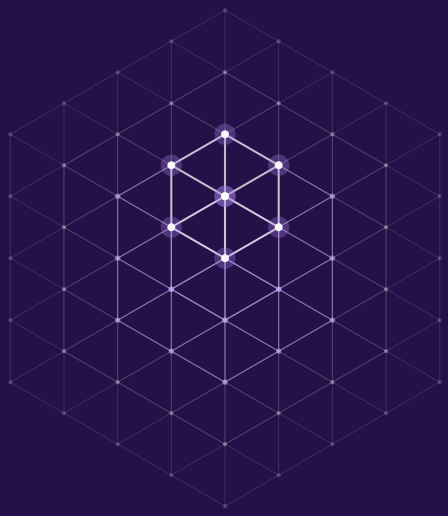
The Lattice Handbook

A Comprehensive Guide to the Lattice Programming Language



Alex Jokela

Tiny Machines Workshop™

Covers Lattice v0.6.4

The Lattice Handbook
A Comprehensive Guide to the Lattice Programming Language

Copyright © 2026 Alex Jokela.
All rights reserved.

ISBN 979-8-9951255-1-8

Covers Lattice v0.6.4.

Second edition, 2026.

Published by Tiny Machines Workshop, LLC
https://tinymachines.io/

Contents

I First Contact 1

1 Hello, Lattice 3

1.1 What Lattice Is and Why It Exists . . . 3

1.1.1 The Phase System . . . 4

1.1.2 Familiar Syntax, Modern Features . . . 5

1.1.3 Batteries Included . . . 5

1.2 Installing Lattice . . . 5

1.2.1 The Quick Path: Homebrew . . . 5

1.2.2 Prerequisites . . . 5

1.2.3 macOS . . . 6

1.2.4 Linux . . . 6

1.2.5 Windows . . . 7

1.2.6 WebAssembly . . . 7

1.2.7 Verifying the Installation . . . 7

1.3 Your First Program: Hello, World! . . . 8

1.3.1 A Slightly More Interesting Program . . . 9

1.3.2 Functions . . . 9

1.3.3 Running the Examples Directory . . . 10

1.4 Running Files vs. the REPL . . . 11

1.4.1 Running from a File . . . 11

1.4.2 The Interactive REPL . . . 12

1.4.3 Passing Arguments to Scripts . . . 13

1.5 A Taste of What Makes Lattice Different . . . 14

1.5.1 The Phase System in Action . . . 14

1.5.2 Pattern Matching . . . 15

1.5.3 Structured Concurrency . . . 15

1.5.4 Closures and Functional Style . . . 16
1.5.5 String Interpolation Everywhere . . . 16
1.5.6 One More Thing: Strict Mode . . . 17
1.6 Exercises . . . 18

2 The REPL Is Your Laboratory 19
2.1 Launching the REPL . . . 19
2.1.1 Persistent State Across Lines . . . 20
2.1.2 Multi-line Input . . . 20
2.2 Tab Completion and Discoverability . . . 21
2.2.1 Keyword and Built-in Completion . . . 22
2.2.2 Constructor Completion . . . 22
2.2.3 What About Method Names? . . . 23
2.3 Defining Functions, Structs, and Enums Interactively . . . 23
2.3.1 Functions . . . 23
2.3.2 Structs . . . 24
2.3.3 Enums . . . 25
2.4 The => Prefix: How the REPL Talks Back . . . 25
2.4.1 The Display Rule . . . 25
2.4.2 The `repr` Format . . . 26
2.4.3 Distinguishing `print` from => . . . 27
2.5 Tips and Tricks for Exploratory Programming . . . 28
2.5.1 Use `typeof()` to Inspect Values . . . 28
2.5.2 Use `phase_of()` to Inspect Phase . . . 28
2.5.3 Build Data Structures Incrementally . . . 29
2.5.4 Test Functions with Edge Cases . . . 29
2.5.5 Use History to Save Typing . . . 30
2.5.6 Explore the Standard Library . . . 30
2.5.7 Error Recovery . . . 31
2.5.8 The Debugging REPL . . . 32
2.5.9 Alternative REPL Backends . . . 33
2.5.10 A Complete Exploratory Session . . . 34
2.6 Exercises . . . 35

3 Values, Types, and the Shape of Things 37
3.1 Integers . . . 37
3.2 Floats . . . 38
3.3 Booleans . . . 39
3.3.1 Truthiness . . . 39
3.4 Strings . . . 40

3.4.1 Double-Quoted Strings . . . 40
3.4.2 String Interpolation . . . 41
3.4.3 Single-Quoted Strings . . . 42
3.4.4 Triple-Quoted Strings . . . 43
3.4.5 Common String Methods . . . 44
3.5 Nil and Unit . . . 45
3.5.1 Nil . . . 45
3.5.2 Unit . . . 46
3.6 Type Checking with `typeof()` . . . 46
3.7 Type Annotations and Function Types . . . 48
3.8 Number Literals in Detail . . . 49
3.8.1 Underscore Separators . . . 50
3.8.2 Hexadecimal Literals . . . 50
3.8.3 No Scientific Notation . . . 50
3.9 Operators . . . 51
3.9.1 Arithmetic Operators . . . 51
3.9.2 Comparison Operators . . . 51
3.9.3 Logical Operators . . . 52
3.9.4 Bitwise Operators . . . 53
3.9.5 Compound Assignment . . . 54
3.9.6 The Nil Coalescing Operator . . . 55
3.9.7 Optional Chaining . . . 55
3.9.8 The Range Operator . . . 56
3.10 Comments . . . 56
3.10.1 Line Comments . . . 56
3.10.2 Block Comments . . . 57
3.10.3 Doc Comments . . . 57
3.11 Putting It All Together . . . 57
3.12 Exercises . . . 59

4 Variables and the Idea of Phase 61
4.1 `let`, `flux`, and `fix` . . . 61
4.1.1 `let` — Inferred Phase . . . 61
4.1.2 `flux` — Explicitly Fluid . . . 62
4.1.3 `fix` — Explicitly Crystal . . . 63
4.1.4 Freeze Protects the Value, Not the Binding . . . 64
4.1.5 The Shorthand Prefixes: `~` and `*` . . . 65
4.1.6 Comparing the Three Keywords . . . 65
4.2 Why Mutability Is a First-Class Concept . . . 66
4.3 Quick Introduction to `freeze()` and `thaw()` . . . 68

4.3.1 `freeze()` — Fluid to Crystal . . . 68
4.3.2 `thaw()` — Crystal to Fluid . . . 69
4.3.3 `clone()` — Independent Copy . . . 70
4.4 `#mode casual` vs. `#mode strict` . . . 71
4.4.1 Casual Mode . . . 71
4.4.2 Strict Mode . . . 72
4.4.3 Inside the Phase Checker . . . 73
4.5 First Encounter with the Phase Philosophy . . . 74
4.5.1 The Lifecycle of Data . . . 74
4.5.2 The Chemistry Metaphor . . . 75
4.5.3 Phase and Concurrency . . . 76
4.5.4 A Preview: The Forge Block . . . 77
4.6 Practical Patterns . . . 77
4.6.1 Accumulate Then Freeze . . . 77
4.6.2 Configuration Objects . . . 78
4.6.3 The Fluid Working Copy . . . 79
4.7 Exercises . . . 80

II The Working Programmer 83

5 Control Flow 85
5.1 if/else as Expressions . . . 85
5.1.1 How the Value Is Determined . . . 86
5.1.2 What Happens Without an else? . . . 87
5.1.3 Nested and Chained Conditions . . . 88
5.1.4 Blocks as Expressions . . . 89
5.2 Loops: while, loop, for . . . 89
5.2.1 while Loops . . . 89
5.2.2 Infinite Loops with loop . . . 91
5.2.3 for Loops and Iteration . . . 92
5.2.4 Nested Loops . . . 94
5.3 break, continue, return . . . 94
5.3.1 break — Exit the Loop . . . 94
5.3.2 continue — Skip to the Next Iteration . . . 95
5.3.3 return — Exit the Function . . . 96
5.4 Ranges . . . 98
5.4.1 Creating Ranges . . . 98
5.4.2 Ranges with Expressions . . . 99
5.4.3 Ranges for Indexing and Slicing . . . 100

5.4.4 Ranges as First-Class Values . . . 100
5.5 The Nil-Coalescing Operator ?? . . . 101
5.5.1 Short-Circuit Evaluation . . . 102
5.5.2 Chaining ?? . . . 103
5.5.3 Combining ?? with Other Expressions . . . 103
5.6 The Pipe Operator . . . 104
5.6.1 How pipe() Works . . . 105
5.6.2 Building Data Pipelines . . . 106
5.6.3 pipe() with Named Functions . . . 107
5.7 Putting It All Together . . . 109
5.8 Exercises . . . 111

6 Functions and Closures 113
6.1 Defining Functions with fn . . . 113
6.1.1 Functions Are Values . . . 114
6.1.2 Function Bodies as Expressions . . . 115
6.1.3 Recursive Functions . . . 116
6.2 Type Annotations on Parameters and Return Types . . . 117
6.2.1 How Runtime Type Checking Works . . . 117
6.2.2 Return Type Checking . . . 118
6.2.3 Common Type Names . . . 118
6.2.4 Spelling Out a Function's Shape . . . 118
6.3 Default Parameter Values and Variadic Parameters . . . 120
6.3.1 Default Parameter Values . . . 120
6.3.2 Variadic Parameters . . . 121
6.3.3 Combining Defaults and Variadics . . . 122
6.4 Closures: `|x| x * 2` . . . 123
6.4.1 Closures with Default and Variadic Parameters . . . 124
6.4.2 Where Closures Shine . . . 125
6.5 Closures Capture Their Environment . . . 126
6.5.1 How Upvalue Capture Works Under the Hood . . . 126
6.5.2 Mutating a Captured Container . . . 128
6.5.3 Closures in Loops . . . 130
6.6 Higher-Order Functions . . . 132
6.6.1 Functions That Return Functions . . . 132
6.6.2 Callbacks and Event Handlers . . . 132
6.6.3 When a Callback Fails . . . 133
6.6.4 Function Composition . . . 135
6.6.5 Building Abstractions with Higher-Order Functions . . . 135
6.7 require and ensure — Contracts on Your Functions . . . 137

6.7.1 Preconditions with require . . . 138
6.7.2 Postconditions with ensure . . . 138
6.7.3 When to Use Contracts . . . 139
6.8 Putting It All Together . . . 140
6.9 Exercises . . . 142

7 Arrays, Maps, Sets, and Friends 143
7.1 Arrays: Creation, Indexing, and Mutation . . . 143
7.1.1 Indexing and Slicing . . . 144
7.1.2 Mutation with push and pop . . . 145
7.1.3 Mutating Nested Elements . . . 146
7.1.4 Essential Non-Closure Methods . . . 147
7.2 Closure-Based Array Methods . . . 149
7.2.1 map — Transform Every Element . . . 149
7.2.2 filter — Keep Matching Elements . . . 149
7.2.3 reduce — Collapse to a Single Value . . . 150
7.2.4 sort_by — Custom Sorting . . . 151
7.2.5 group_by — Categorize Elements . . . 151
7.2.6 More Closure Methods . . . 153
7.2.7 When a Callback Fails . . . 154
7.3 Non-Closure Array Methods Reference . . . 155
7.4 The Spread Operator 156
7.5 Maps . . . 157
7.5.1 Map Methods . . . 158
7.5.2 Updating Values In Place . . . 160
7.5.3 Maps as Lightweight Objects . . . 161
7.6 Sets . . . 162
7.6.1 Set Operations . . . 162
7.6.2 Subset and Superset Testing . . . 164
7.6.3 Converting Between Sets and Arrays . . . 164
7.7 Tuples . . . 164
7.7.1 Accessing Tuple Elements . . . 165
7.7.2 Tuples vs. Arrays . . . 166
7.8 Choosing the Right Collection . . . 167
7.9 Exercises . . . 169

8 Strings in Depth 171
8.1 Interpolation, Escaping, and Raw Strings . . . 171
8.1.1 String Interpolation . . . 171
8.1.2 Escape Sequences . . . 172

8.1.3 Single-Quoted Strings . . . 173
8.1.4 Triple-Quoted Strings . . . 173
8.2 Core String Methods . . . 174
8.2.1 split — Break a String Apart . . . 174
8.2.2 trim, trim_start, trim_end . . . 175
8.2.3 replace . . . 176
8.2.4 contains, starts_with, ends_with . . . 176
8.2.5 count . . . 176
8.3 Case Transformations . . . 177
8.3.1 Naming Convention Converters . . . 177
8.4 Characters, Bytes, and Substrings . . . 179
8.4.1 chars() and bytes() . . . 179
8.4.2 len(), substring(), and index_of() . . . 180
8.4.3 More String Utilities . . . 181
8.5 Binary-Safe Strings . . . 182
8.5.1 Getting a NUL Into a String . . . 183
8.5.2 Every Method Honors the Length . . . 184
8.5.3 Where the Old Assumption Survives . . . 185
8.6 Character Codes: ord and chr . . . 186
8.6.1 ord() and chr() . . . 186
8.6.2 Building a Caesar Cipher . . . 188
8.7 Regular Expressions . . . 188
8.7.1 regex_match — Test a Pattern . . . 189
8.7.2 regex_find_all — Extract All Matches . . . 189
8.7.3 regex_replace — Pattern-Based Replacement . . . 190
8.7.4 Regex Flags . . . 191
8.8 Putting It All Together . . . 191
8.9 Exercises . . . 193

9 Pattern Matching 195
9.1 The `match` Expression . . . 196
9.1.1 Arm Bodies: Expressions and Blocks . . . 197
9.1.2 First Match Wins . . . 197
9.2 Literal Patterns . . . 198
9.3 Wildcard and Binding Patterns . . . 199
9.3.1 The Wildcard _ . . . 199
9.3.2 Binding Patterns . . . 199
9.4 Range Patterns . . . 200
9.5 Guards . . . 202
9.5.1 Guards with Other Pattern Types . . . 203

9.6 Array Destructuring Patterns . . . 204
9.6.1 Rest Patterns . . . 205
9.6.2 Array Patterns and Guards . . . 206
9.7 Struct Destructuring Patterns . . . 207
9.8 Enum Variant Patterns . . . 208
9.8.1 Variants Without Payloads . . . 210
9.8.2 Nested Enum Patterns . . . 210
9.8.3 Enum Matching with Guards . . . 212
9.9 Exhaustiveness Checking . . . 212
9.9.1 How the Checker Works . . . 213
9.9.2 Enum Exhaustiveness . . . 213
9.9.3 Boolean Exhaustiveness . . . 214
9.9.4 Limitations of the Checker . . . 215
9.10 Phase-Qualified Patterns . . . 216
9.11 Putting It All Together . . . 216
9.12 How Match Compiles . . . 219
9.12.1 The Hidden Scrutinee Slot . . . 219
9.12.2 Compilation by Pattern Type . . . 220
9.12.3 Match in Expression Position . . . 221
9.13 Common Patterns and Idioms . . . 222
9.13.1 Replacing Long if/else Chains . . . 222
9.13.2 Extracting from Nested Structures . . . 223
9.13.3 State Machines . . . 224
9.13.4 The Option Pattern . . . 225
9.14 Exercises . . . 226

10 Error Handling 229
10.1 Error Values: `error()` and `is_error()` . . . 230
10.2 The Result Type . . . 232
10.2.1 Working with Results . . . 233
10.3 try/catch . . . 235
10.3.1 try/catch Is an Expression . . . 237
10.3.2 The Error Map . . . 237
10.3.3 What Can Be Caught? . . . 240
10.4 The ? Operator . . . 242
10.4.1 How ? Works . . . 243
10.4.2 Chaining ? . . . 244
10.5 panic() . . . 245
10.6 defer . . . 246
10.6.1 Execution Order: LIFO . . . 247

10.6.2 Scope-Aware Execution . . . 248
10.6.3 Defer and Errors . . . 248
10.6.4 Defer and Return Values . . . 249
10.6.5 How Defer Compiles . . . 250
10.7 Contracts Revisited . . . 250
10.7.1 require: Preconditions . . . 250
10.7.2 ensure: Postconditions . . . 251
10.7.3 Combining Contracts with try/catch . . . 251
10.8 Assertions . . . 252
10.9 The `try_fn` Helper . . . 253
10.10 Designing for Graceful Failure . . . 254
10.10.1 Choosing the Right Tool . . . 254
10.10.2 The Error Boundary Pattern . . . 255
10.10.3 Fail Fast, Recover High . . . 255
10.11 Exercises . . . 256

III The Phase System 259

11 Phases Explained 261
11.1 The Philosophy: Mutability as a Material Property . . . 261
11.2 `flux` (Fluid) — Mutable, Alive, in Motion . . . 263
11.2.1 Fluid Collections . . . 263
11.2.2 Fluid Reassignment . . . 264
11.3 `fix` (Crystal) — Immutable, Hardened, Permanent . . . 265
11.3.1 Why Crystal Is More Than "const" . . . 265
11.3.2 Crystal Values and Memory . . . 266
11.3.3 Freezing Protects the Value, Not the Name . . . 266
11.4 `let` — Phase Inference . . . 267
11.4.1 The Relationship Between `let` and Strict Mode . . . 269
11.5 `freeze()`, `thaw()`, `clone()` in Depth . . . 269
11.5.1 `freeze()` — Crystallization . . . 269
11.5.2 `thaw()` — Melting a Crystal . . . 272
11.5.3 `clone()` — Independent Copy . . . 275
11.6 Advanced Phase Operations . . . 276
11.6.1 `forge` — Scoped Construction . . . 276
11.6.2 `anneal` — Transform a Crystal . . . 277
11.6.3 `crystallize` and `borrow` — Scoped Phase Changes . . . 277
11.6.4 `sublimate` — The Strictest Lock . . . 278
11.6.5 `freeze(x) except [...]` — Partial Freezing . . . 280

11.7 When and Why to Freeze Values . . . 282
11.7.1 Sharing Data Between Threads . . . 282
11.7.2 API Boundaries . . . 283
11.7.3 Snapshot Semantics . . . 283
11.7.4 Performance Optimization . . . 284
11.7.5 Correctness Guarantees . . . 284
11.8 Crystal Sharing — How Frozen Values Are Stored . . . 285
11.8.1 What Shares and What Does Not . . . 285
11.8.2 How Operations Behave Since v0.6.0 . . . 286
11.8.3 Concurrency Interaction . . . 286
11.8.4 Switches and Debugging Modes . . . 287
11.8.5 Performance . . . 287
11.8.6 Known Limitation . . . 288
11.9 Phase Errors and What They Tell You . . . 288
11.9.1 Mutating a Crystal Value . . . 288
11.9.2 Assigning to a Crystal Binding . . . 289
11.9.3 Strict Mode: `let` Not Allowed . . . 290
11.9.4 Strict Mode: Phase Mismatch . . . 290
11.9.5 Strict Mode: Freezing an Already Crystal Value . . . 291
11.9.6 Strict Mode: Thawing an Already Fluid Value . . . 291
11.9.7 Strict Mode: Fluid Values in `spawn` . . . 292
11.9.8 Annotation Violations . . . 292
11.10 Querying Phase at Runtime . . . 293
11.11 Summary: The Phase Lifecycle . . . 294
11.12 Exercises . . . 295

12 Memory and Arenas **297**
12.1 The FluidHeap: Mark-Sweep GC for Mutable Values . . . 297
12.1.1 How the FluidHeap Works . . . 298
12.1.2 Mark-Sweep Collection . . . 298
12.1.3 The GC Threshold and Adaptive Growth . . . 299
12.2 The CrystalRegion: Arena-Backed Storage for Immutable Values . . . 300
12.2.1 What Is an Arena? . . . 300
12.2.2 Region Structure . . . 300
12.2.3 How Freeze Moves Data . . . 301
12.2.4 Region Lifetime: Refcounts, Not Collection . . . 302
12.2.5 The RegionManager . . . 303
12.3 Two Kinds of Crystal . . . 304
12.3.1 Turning Sharing Off . . . 304
12.3.2 Crystallized Maps . . . 305

12.4 The Ephemeral BumpArena: Short-Lived Temporaries . . . 306
12.4.1 How the BumpArena Works . . . 306
12.4.2 The Reset Cycle . . . 306
12.5 How Freeze Moves a Value from Heap to Arena . . . 307
12.5.1 Step by Step . . . 307
12.5.2 Memory Layout Comparison . . . 309
12.6 String Interning and Why It Matters . . . 309
12.6.1 How Interning Works . . . 309
12.6.2 What Gets Interned . . . 310
12.6.3 Why Interning Matters for Phase . . . 310
12.7 The DualHeap: Putting It All Together . . . 311
12.8 GC Flags: Tuning the Collector . . . 311
12.8.1 `-gc`: Enable the Garbage Collector . . . 311
12.8.2 `-gc-stress`: Collect on Every Allocation . . . 312
12.8.3 `-gc-incremental`: Spread Work Across Safe Points . . . 312
12.8.4 GC Statistics . . . 313
12.9 Special Region IDs . . . 313
12.10 Memory Best Practices . . . 314
12.11 Exercises . . . 315

13 Alloys and Reactive Bonds **317**
13.1 Alloy Structs: Per-Field Phase Declarations . . . 317
13.1.1 How Alloys Work Under the Hood . . . 319
13.1.2 Partial Freeze with `except` . . . 320
13.2 Designing Data with Mixed Mutability . . . 322
13.2.1 Identity + State Pattern . . . 322
13.2.2 Configuration + Runtime Pattern . . . 323
13.2.3 Append-Only Pattern . . . 324
13.3 Reactive Bonds: `bond()`, `unbond()`, `react()`, `unreact()` . . . 326
13.3.1 Reactions: Responding to Phase Changes . . . 326
13.3.2 Removing Reactions . . . 328
13.3.3 Bonds: Phase Dependency Relationships . . . 328
13.3.4 Removing Bonds . . . 330
13.4 Bond Strategies: Mirror, Inverse, Gate . . . 330
13.4.1 Mirror: Freeze Follows Freeze . . . 331
13.4.2 Inverse: Freeze Releases . . . 331
13.4.3 Gate: Freeze Requires Freeze . . . 332
13.5 Pressure: `pressurize()`, `depressurize()`, `pressure_of()` . . . 332
13.5.1 Pressure Modes . . . 333
13.5.2 Querying and Removing Pressure . . . 335

13.5.3 Implementation Details . . . 335
13.6 Seeds: `seed()`, `unseed()` . . . 335
13.6.1 Seeds and `grow()` . . . 337
13.6.2 Removing Seeds Manually . . . 338
13.6.3 Multiple Seeds . . . 338
13.7 Building Reactive Data Flows . . . 338
13.7.1 Example: Configuration Validation Pipeline . . . 339
13.7.2 Example: Sensor Data with Pressure Guards . . . 341
13.7.3 Example: Multi-Stage Initialization . . . 342
13.8 Exercises . . . 343

14 Strict Mode and Static Phase Checking **345**
14.1 `#mode strict` — What Changes . . . 345
14.1.1 Consume on Freeze . . . 347
14.2 The Static Phase Checker . . . 348
14.2.1 Architecture . . . 348
14.2.2 Scope Tracking . . . 349
14.2.3 Expression Phase Inference . . . 349
14.2.4 Statement Checking . . . 350
14.2.5 The Spawn Checker . . . 350
14.3 Phase Annotations: `@fluid`, `@crystal` . . . 351
14.3.1 Annotation Syntax . . . 351
14.3.2 What Annotations Do . . . 352
14.3.3 Annotations on Functions . . . 353
14.4 Phase Constraints: `(~| *)`, `(flux| fix)` . . . 353
14.4.1 Constraint Syntax . . . 354
14.4.2 Constraint Checking . . . 355
14.5 Phase-Dependent Function Overloading . . . 356
14.5.1 How Overload Resolution Works . . . 357
14.5.2 Overloading in Strict Mode . . . 358
14.5.3 Registration and Chaining . . . 358
14.6 Designing APIs That Are Phase-Aware . . . 359
14.6.1 Produce Crystal, Accept Any . . . 359
14.6.2 Phase-Specific Behavior via Overloading . . . 360
14.6.3 Guard Clauses with `phase_of()` . . . 361
14.6.4 Strict Mode in Library Code . . . 361
14.6.5 Documenting Phase Expectations . . . 362
14.7 Putting It All Together: A Strict-Mode Example . . . 362
14.8 Exercises . . . 364

IV Structures and Abstractions 365

15 Structs 367
15.1 Defining and Instantiating Structs . . . 367
15.1.1 Nested Structs . . . 369
15.1.2 Structs with Typed Fields . . . 369
15.2 Field Access and Mutation . . . 370
15.2.1 Mutation and the Phase System . . . 370
15.2.2 Optional Field Access . . . 373
15.2.3 Passing Structs to Functions . . . 373
15.2.4 Structs in Collections . . . 374
15.3 Methods via Traits: `impl` Blocks . . . 375
15.3.1 Multiple Traits per Struct . . . 376
15.3.2 Methods in Loops . . . 377
15.4 Callable Struct Fields . . . 378
15.4.1 Stateful Callable Fields . . . 379
15.4.2 How the Runtime Resolves Calls . . . 382
15.5 Custom Equality with `eq` . . . 382
15.5.1 Overriding Equality with an `eq` Field . . . 382
15.5.2 When Custom Equality Matters . . . 383
15.6 Struct Reflection . . . 384
15.6.1 `struct_name()` . . . 384
15.6.2 `struct_fields()` . . . 385
15.6.3 `struct_to_map()` . . . 385
15.6.4 `struct_from_map()` . . . 386
15.6.5 A Practical Reflection Example . . . 386
15.7 Struct Destructuring . . . 387
15.8 Pass-by-Value Semantics . . . 388
15.8.1 Implications for Function Calls . . . 389
15.8.2 Why Pass-by-Value? . . . 390
15.8.3 Explicit Copies with `clone()` . . . 390
15.8.4 When Copies Get Expensive . . . 391
15.9 A Larger Example: Entity Component System . . . 392
15.10 Exercises . . . 394

16 Enums 397
16.1 Defining Enums With and Without Payloads . . . 397
16.1.1 Variants with Payloads . . . 398
16.2 Constructing Variants . . . 399
16.2.1 Enums Returned from Functions . . . 400

16.2.2 Enums in Collections . . . 402
16.2.3 Runtime Introspection Methods . . . 402
16.3 Matching on Enums . . . 403
16.3.1 Matching Unit Variants . . . 404
16.3.2 Destructuring Payloads . . . 405
16.3.3 Multi-Element Payloads . . . 406
16.3.4 Using Destructured Values in Computations . . . 407
16.3.5 Block Bodies in Arms . . . 407
16.3.6 Ignoring Payload with Wildcards . . . 408
16.3.7 Nested Match Expressions . . . 408
16.4 Using Enums for State Machines . . . 409
16.4.1 Driving State Transitions . . . 410
16.4.2 A Parser Token Example . . . 412
16.5 Option-Like and Result-Like Patterns . . . 414
16.5.1 The Option Pattern . . . 414
16.5.2 The Result Pattern . . . 416
16.5.3 Chaining Results . . . 416
16.6 Exhaustiveness Warnings . . . 417
16.6.1 How It Works . . . 418
16.6.2 Boolean Exhaustiveness . . . 419
16.6.3 Non-Enum Matches . . . 419
16.6.4 Make the Wildcard Arm Loud . . . 420
16.7 Recursive Enums . . . 420
16.7.1 An Expression Evaluator . . . 422
16.8 Practical Patterns . . . 422
16.8.1 Enums and Loops . . . 422
16.8.2 Enums as Function Return Types . . . 423
16.8.3 Under the Hood . . . 424
16.9 Exercises . . . 425

17 Traits and Impl Blocks 427
17.1 Declaring Traits . . . 427
17.1.1 Multi-Method Traits . . . 428
17.1.2 Traits with Extra Parameters . . . 428
17.2 Implementing Traits for Structs . . . 429
17.2.1 How `self` Works . . . 429
17.2.2 Accessing Fields through `self` . . . 430
17.2.3 Method Chaining . . . 431
17.3 Multiple Impl Blocks per Struct . . . 433
17.3.1 How Method Registration Works . . . 435

17.4 Designing with Interfaces . . . 435
17.4.1 The Same Trait on Different Types . . . 435
17.4.2 Traits for Separation of Concerns . . . 437
17.4.3 Naming Conventions . . . 438
17.5 Trait-Driven Polymorphism . . . 438
17.5.1 Heterogeneous Collections . . . 438
17.5.2 Methods in Expressions and Conditionals . . . 440
17.5.3 Methods in Loops . . . 440
17.5.4 Traits on Enums . . . 441
17.5.5 A Complete Polymorphism Example . . . 442
17.6 Under the Hood . . . 444
17.6.1 Compilation . . . 444
17.6.2 Dispatch Performance . . . 444
17.6.3 Self Cloning . . . 444
17.7 Exercises . . . 445

18 Generics: The Syntax Reserved for the Future 447
18.1 Generic Functions and the Runtime Type Checker . . . 448
18.1.1 Why This Happens . . . 448
18.1.2 Exactly Where the Line Falls . . . 449
18.2 Generic Structs . . . 451
18.2.1 Multi-Parameter Structs . . . 452
18.2.2 Realistic Generic Structs . . . 452
18.2.3 Generic Structs with Traits . . . 453
18.3 Generic Enums and Trait Implementations . . . 454
18.3.1 Generic Result . . . 455
18.3.2 Recursive Generic Enums . . . 456
18.4 Type Expressions . . . 456
18.4.1 Named Types . . . 457
18.4.2 Parameterised Named Types . . . 457
18.4.3 Array Types . . . 457
18.4.4 Function Types . . . 458
18.4.5 Combining Type Expressions . . . 459
18.4.6 Phase Annotations in Types . . . 460
18.5 Practical Patterns with Generics . . . 461
18.5.1 The Wrapper Pattern . . . 461
18.5.2 The Builder Pattern . . . 461
18.5.3 Higher-Order Generic Functions . . . 462
18.5.4 Generic Option Utilities . . . 464
18.5.5 Combining Generics, Enums, and Traits . . . 464

18.6 The State of Generics in Lattice . . . 466
18.7 Exercises . . . 467

V Concurrency 469

19 Structured Concurrency with scope and spawn 471
19.1 The scope { spawn { . . . } } Model . . . 471
19.1.1 What Goes Where . . . 472
19.1.2 Multiple Spawns . . . 473
19.2 Why Structured Concurrency Matters . . . 474
19.2.1 The Problem with Unstructured Threads . . . 474
19.2.2 Lattice's Guarantee . . . 475
19.3 Joining Semantics—No Orphaned Tasks . . . 476
19.3.1 Nested Scopes . . . 477
19.3.2 Scopes in Loops . . . 478
19.4 Sharing Data Between Spawned Tasks . . . 479
19.4.1 Communicating Through Channels . . . 480
19.4.2 Communicating Through Ref . . . 482
19.5 The Sublimate Phase for Thread-Safe Values . . . 482
19.5.1 Using sublimate . . . 483
19.5.2 Sublimate vs. Freeze . . . 484
19.5.3 Sublimating Complex Structures . . . 486
19.6 Practical Examples . . . 487
19.6.1 Parallel Computation: Fan-Out / Fan-In . . . 487
19.6.2 Parallel String Processing . . . 488
19.6.3 Nested Parallelism: Two-Level Fan-Out . . . 489
19.6.4 Using a Cancellation Flag . . . 490
19.6.5 Scope with Error Handling . . . 492
19.7 Under the Hood . . . 492
19.7.1 Compilation . . . 492
19.7.2 Execution . . . 493
19.7.3 What the Runtime Guarantees . . . 493
19.8 Exercises . . . 494

20 Channels and select 497
20.1 Channel::new(), send(), recv(), close() . . . 497
20.1.1 send() — Putting Values In . . . 498
20.1.2 Send Eligibility Rules . . . 499
20.1.3 recv() — Taking Values Out . . . 501

20.1.4 What Happens When You `recv()` from a Closed, Empty Channel? . . . 502
20.1.5 `close()` — Signaling Completion . . . 503
20.1.6 Channel Lifecycle . . . 504
20.2 Producer-Consumer Patterns . . . 505
20.2.1 Single Producer, Single Consumer . . . 505
20.2.2 Multiple Producers, Single Consumer . . . 506
20.2.3 Single Producer, Multiple Consumers . . . 507
20.3 `select` with Multiple Channels . . . 508
20.3.1 Anatomy of a `select` Arm . . . 509
20.3.2 Fairness . . . 510
20.3.3 Select in a Loop . . . 512
20.4 `default` and `timeout` Arms . . . 514
20.4.1 The `default` Arm . . . 514
20.4.2 The `timeout` Arm . . . 516
20.4.3 Combining `default` and `timeout` . . . 517
20.4.4 Select with All Channels Closed . . . 518
20.5 Building Pipelines with Channels . . . 518
20.5.1 A Two-Stage Pipeline . . . 519
20.5.2 A Three-Stage Pipeline . . . 519
20.5.3 Fan-Out Pipeline . . . 521
20.6 Async Iterators: `async_iter`, `async_map`, `async_filter` . . . 521
20.6.1 `async_iter` — Channel-Backed Iterators . . . 522
20.6.2 `async_map` — Transform an Async Stream . . . 523
20.6.3 `async_filter` — Filter an Async Stream . . . 524
20.6.4 Composing Async Operations . . . 524
20.7 Under the Hood: `select` Implementation . . . 525
20.7.1 Compilation . . . 525
20.7.2 Execution . . . 526
20.8 Exercises . . . 526

21 Concurrency Patterns and Pitfalls 529
21.1 `Ref` for Shared Mutable State . . . 529
21.1.1 `Ref::new()`, `.get()`, `.set()` . . . 529
21.1.2 Ref Proxying . . . 530
21.2 When to Use Channels vs. `Ref` . . . 531
21.2.1 The Core Trade-Off . . . 532
21.3 Deadlock Avoidance . . . 533
21.3.1 Classic Channel Deadlock . . . 533
21.3.2 Rules for Avoiding Deadlocks . . . 533
21.3.3 Deadlock-Free Pattern: The Coordinator . . . 535

21.4 Common Patterns . . . 535
21.4.1 Worker Pool . . . 535
21.4.2 Broadcast . . . 537
21.4.3 Request-Reply . . . 538
21.5 Testing Concurrent Code . . . 540
21.5.1 Deterministic Tests with Channels . . . 540
21.5.2 Stress Testing with Repeated Runs . . . 541
21.5.3 Testing with Timeouts . . . 541
21.5.4 Testing Error Propagation . . . 542
21.5.5 Rules of Thumb for Concurrent Tests . . . 544
21.6 Exercises . . . 545

VI The Standard Library **547**

22 Files, Paths, and the Filesystem **549**
22.1 read_file, write_file, append_file . . . 549
22.1.1 A Complete Read-Modify-Write Cycle . . . 551
22.2 The fs Module . . . 551
22.2.1 Checking Existence: file_exists, is_file, is_dir . . . 552
22.2.2 Listing Directories: list_dir . . . 552
22.2.3 Creating and Removing Directories: mkdir, rmdir . . . 553
22.2.4 Globbing: glob . . . 554
22.2.5 File Metadata: stat and file_size . . . 554
22.2.6 Copying, Renaming, and Deleting . . . 555
22.2.7 Permissions and Canonical Paths . . . 556
22.3 The path Module . . . 556
22.3.1 Joining Paths: path_join . . . 557
22.3.2 Dissecting Paths: path_dir, path_base, path_ext . . . 557
22.3.3 Building a File Organizer . . . 558
22.4 Temporary Files and Directories . . . 560
22.4.1 A Safe Processing Pipeline . . . 561
22.5 Working with Buffers . . . 562
22.5.1 Creating Buffers . . . 562
22.5.2 Reading and Writing Binary Files . . . 563
22.5.3 Indexing and Mutation . . . 564
22.5.4 The push Method and Dynamic Growth . . . 565
22.5.5 Typed Read and Write Methods . . . 565
22.5.6 Slicing, Filling, and Converting . . . 568
22.5.7 Parsing a Binary File Format . . . 568

23 JSON, TOML, YAML, and CSV **573**
23.1 json_parse / json_stringify . . . 573
23.1.1 Parsing JSON . . . 573
23.1.2 Stringifying to JSON . . . 575
23.1.3 A Practical Example: Reading and Writing JSON Files . . . 577
23.2 toml_parse / toml_stringify . . . 577
23.2.1 Parsing TOML . . . 578
23.2.2 Stringifying to TOML . . . 580
23.3 yaml_parse / yaml_stringify . . . 580
23.3.1 Parsing YAML . . . 581
23.3.2 Stringifying to YAML . . . 583
23.4 csv_parse / csv_stringify . . . 583
23.4.1 Parsing CSV . . . 584
23.4.2 Working with Headers . . . 585
23.4.3 Stringifying to CSV . . . 586
23.5 Strictness, Limits, and Hostile Input . . . 587
23.5.1 JSON Says No . . . 587
23.5.2 Depth Limits . . . 588
23.5.3 Other Guarded Surfaces . . . 589
23.6 Round-Tripping Data Between Formats . . . 589
23.6.1 CSV to JSON . . . 590
23.7 Building Configuration-Driven Programs . . . 592
23.7.1 The Configuration File . . . 592
23.7.2 The Analyzer . . . 594
23.7.3 Using the dotenv Library . . . 595

24 Networking and HTTP **597**
24.1 The http Module . . . 597
24.1.1 Simple GET Requests . . . 598
24.1.2 POST Requests . . . 600
24.1.3 The General-Purpose http_request . . . 601
24.1.4 Error Handling in HTTP . . . 602
24.1.5 What the Client Refuses to Send . . . 604
24.2 The net Module: TCP Networking . . . 605
24.2.1 Connecting to a Server . . . 605
24.2.2 Building a TCP Server . . . 607
24.2.3 TCP Function Reference . . . 608
24.3 TLS: Secure Connections . . . 608
24.3.1 Making a TLS Connection . . . 609
24.3.2 TLS Function Reference . . . 610

24.4 Building a Simple HTTP Server . . . 610
24.4.1 A Server from Scratch . . . 610
24.4.2 The http_server Library . . . 612
24.4.3 A Complete REST API . . . 617
24.5 URL Encoding and Decoding . . . 619

25 Time, Crypto, OS, and Everything Else 623
25.1 The time Module . . . 623
25.1.1 Getting the Current Time . . . 623
25.1.2 Sleeping . . . 624
25.1.3 Formatting Timestamps . . . 624
25.1.4 Parsing Timestamps . . . 625
25.1.5 Extracting Date Components . . . 625
25.1.6 Date Arithmetic . . . 626
25.1.7 The datetime Functions . . . 627
25.2 The crypto Module . . . 629
25.2.1 Hashing: sha256, sha512, md5 . . . 629
25.2.2 HMAC: hmac_sha256 . . . 630
25.2.3 Base64 Encoding and Decoding . . . 631
25.2.4 Random Bytes . . . 632
25.3 uuid() . . . 633
25.4 The os Module . . . 634
25.4.1 Environment Variables: env, env_set, env_keys . . . 634
25.4.2 Command-Line Arguments: args() . . . 635
25.4.3 System Information: platform, cwd, hostname, pid . . . 636
25.4.4 Executing External Commands: exec and shell . . . 636
25.4.5 exec_argv: Running a Program Without a Shell . . . 638
25.5 The math Module . . . 641
25.5.1 Rounding and Absolute Value . . . 642
25.5.2 Powers, Roots, and Logarithms . . . 642
25.5.3 Min, Max, Clamp, and Lerp . . . 643
25.5.4 Trigonometry . . . 644
25.5.5 Random Numbers . . . 645
25.5.6 Other Math Functions . . . 646
25.5.7 Where the Numbers Stop . . . 646
25.6 The regex Module . . . 647
25.6.1 Matching: regex_match . . . 648
25.6.2 Finding All Matches: regex_find_all . . . 649
25.6.3 Replacing: regex_replace . . . 651
25.6.4 Regex Size Caps . . . 651

25.6.5 Practical Example: Log Parser . . . 652
25.7 The progress Module . . . 654

VII Iterators and Functional Style 657

26 Lazy Iterators 659
26.1 The Iterator Protocol . . . 659
26.1.1 What Can You Iterate Over? . . . 660
26.1.2 Using Iterators in For Loops . . . 661
26.2 Sources: Where Iterators Come From . . . 661
26.2.1 iter() — From Collections . . . 661
26.2.2 range_iter() — Lazy Integer Sequences . . . 662
26.2.3 repeat_iter() — The Same Value, Over and Over . . . 662
26.3 Transformers: Reshaping the Stream . . . 663
26.3.1 .map() — Transform Each Element . . . 663
26.3.2 .filter() — Keep Only What Matters . . . 664
26.3.3 .take() — Limit the Count . . . 664
26.3.4 .skip() — Jump Ahead . . . 665
26.3.5 .enumerate() — Track the Index . . . 666
26.3.6 .zip() — Walk Two Iterators in Lockstep . . . 666
26.3.7 Chaining Transformers . . . 667
26.4 Consumers: Triggering the Work . . . 668
26.4.1 .collect() — Materialize into an Array . . . 668
26.4.2 .reduce() — Fold into a Single Value . . . 668
26.4.3 .any() and .all() — Short-Circuit Boolean Tests . . . 669
26.4.4 .count() — How Many? . . . 670
26.4.5 Summary of Consumers . . . 670
26.5 When a Callback Fails . . . 671
26.6 Building Your Own Iterators . . . 672
26.6.1 The Sequence Protocol . . . 673
26.6.2 Building a Fibonacci Iterator . . . 673
26.6.3 Custom Sequences from Scratch . . . 674
26.6.4 A Practical Example: Paginated API Results . . . 675
26.6.5 Composing fn Module Sequences . . . 676
26.7 Why Laziness Matters . . . 677
26.7.1 Memory Efficiency . . . 677
26.7.2 Computation Efficiency . . . 678
26.7.3 Infinite Sequences . . . 679
26.7.4 When to Stay Eager . . . 680

26.8 Under the Hood . . . 680
26.9 Exercises . . . 680

27 Functional Programming in Lattice **683**
27.1 pipe(), compose(), and identity() . . . 683
27.1.1 pipe() — Threading Values Through Functions . . . 683
27.1.2 compose() — Building New Functions from Old . . . 685
27.1.3 identity() — The Do-Nothing Function . . . 687
27.2 The fn Standard Library Module . . . 688
27.2.1 Function Composition Utilities . . . 689
27.2.2 apply_n — Repeated Application . . . 689
27.2.3 thread() — Module-Level Piping . . . 690
27.2.4 Collection Utilities . . . 690
27.2.5 The Result Type . . . 692
27.3 Closures That Accumulate . . . 693
27.4 Currying and Partial Application Patterns . . . 695
27.4.1 curry() — One Argument at a Time . . . 695
27.4.2 partial() — Bind Some Arguments Now . . . 696
27.4.3 Currying with Iterators . . . 697
27.5 When Functional Style Shines in Lattice . . . 698
27.5.1 Data Transformation Pipelines . . . 698
27.5.2 Callback-Heavy APIs . . . 699
27.5.3 Configuration and Strategy Patterns . . . 700
27.5.4 When Imperative Style Is Better . . . 701
27.6 Putting It All Together . . . 702
27.7 Exercises . . . 704

VIII Modules, Packages, and Project Structure **705**

28 Modules and Imports **707**
28.1 Your First Import . . . 707
28.2 Import Styles . . . 708
28.2.1 Aliased Import: `import ... as` . . . 708
28.2.2 Selective Import: `import {...} from` . . . 709
28.2.3 Bare Import . . . 710
28.3 The Export System . . . 711
28.3.1 Explicit Exports . . . 711
28.3.2 Exporting Structs and Enums . . . 712
28.3.3 What Gets Filtered Out . . . 713

28.3.4 Re-exporting from Other Modules . . . 714
28.4 Built-in Modules . . . 714
28.5 Module Resolution . . . 716
28.5.1 The Resolution Order . . . 717
28.5.2 Where "Relative" Is Relative To . . . 717
28.5.3 Package Resolution Strategies . . . 718
28.5.4 Relative and Nested Imports . . . 719
28.5.5 Module Caching . . . 721
28.6 Organizing Multi-File Projects . . . 723
28.6.1 A Minimal Project . . . 723
28.6.2 Growing Beyond One File . . . 723
28.6.3 Facade Modules . . . 725
28.6.4 Namespacing with Aliased Imports . . . 728
28.6.5 Scoped Imports . . . 728
28.7 Under the Hood: How Imports Work . . . 729
28.7.1 Parsing . . . 729
28.7.2 Compilation . . . 730
28.7.3 Runtime Execution . . . 730
28.8 Exercises . . . 732

29 The Package Manager **733**
29.1 Initializing a Project with `clat init` . . . 733
29.2 The `lattice.toml` Manifest . . . 734
29.2.1 The `[package]` Section . . . 735
29.2.2 The `[dependencies]` Section . . . 735
29.3 Adding, Removing, and Installing Packages . . . 736
29.3.1 `clat add` . . . 736
29.3.2 Git Dependencies . . . 737
29.3.3 `clat install` . . . 738
29.3.4 `clat remove` . . . 739
29.4 Semantic Versioning and Dependency Resolution . . . 740
29.4.1 Version Constraints . . . 740
29.4.2 Caret vs. Tilde . . . 740
29.4.3 How Resolution Works . . . 741
29.5 The Lock File and Reproducible Builds . . . 742
29.5.1 What the Lock File Looks Like . . . 742
29.5.2 Checksums and Integrity Verification . . . 743
29.5.3 How the Lock File Ensures Reproducibility . . . 743
29.5.4 When Lock Entries Are Updated . . . 744
29.6 The Package Registry and Cache . . . 744

29.6.1 The Lattice Registry . . . 744
29.6.2 Overriding the Registry . . . 744
29.6.3 The Global Package Cache . . . 745
29.6.4 The `lat_modules/` Directory . . . 746
29.7 Dependency Graphs and Cycle Detection . . . 747
29.7.1 How the Graph Is Built . . . 747
29.7.2 Cycle Detection . . . 748
29.8 Trusting Your Dependencies . . . 749
29.8.1 Manifest Fields Are Validated Before Use . . . 749
29.8.2 No Shell, Ever . . . 750
29.9 Under the Hood: The Install Pipeline . . . 751
29.10 Putting It All Together . . . 752
29.11 Exercises . . . 754

IX Tooling and Developer Experience 757

30 The Formatter and Doc Generator 759
30.1 `clat fmt` — Formatting Your Code . . . 759
30.1.1 One File at a Time . . . 761
30.1.2 Check Mode: CI-Friendly Verification . . . 761
30.1.3 Reading from Standard Input . . . 762
30.1.4 Doc Comments Survive, and Formatting Is Idempotent . . . 762
30.2 What the Formatter Does Not Do: `-width` . . . 763
30.3 `///` Doc Comments . . . 765
30.3.1 Documenting All Declaration Types . . . 765
30.3.2 Module-Level Doc Comments . . . 766
30.3.3 Doc Comment Conventions . . . 767
30.4 `clat doc` — Generating Documentation . . . 768
30.4.1 Documenting a Directory . . . 770
30.4.2 Writing Output to Files . . . 771
30.5 Output Formats: Markdown, JSON, HTML . . . 772
30.5.1 Markdown (Default) . . . 772
30.5.2 Plain Text . . . 772
30.5.3 JSON . . . 772
30.5.4 HTML . . . 773
30.5.5 The `-doc-format` Flag . . . 774
30.5.6 How Extraction Works Under the Hood . . . 774
Exercises . . . 775
What's Next . . . 775

31 Testing 777
31.1 `test "name" { body }` — Inline Tests . . . 777
31.1.1 Tests Are Top-Level Declarations . . . 779
31.1.2 Tests Can Use Everything in the File . . . 780
31.1.3 What Happens During Normal Execution? . . . 781
31.2 `clat test` — Running the Test Suite . . . 781
31.2.1 Directory Discovery . . . 782
31.2.2 Filtering Tests by Name . . . 783
31.2.3 Verbose and Summary Modes . . . 784
31.2.4 Full CLI Reference . . . 785
31.3 Assert Functions . . . 785
31.3.1 `assert(condition, message?)` . . . 785
31.3.2 `assert_eq(actual, expected)` . . . 786
31.3.3 `assert_ne(actual, expected)` . . . 788
31.3.4 `assert_true(value)` and `assert_false(value)` . . . 788
31.3.5 `assert_type(value, type_name)` . . . 789
31.3.6 `assert_contains(haystack, needle)` . . . 789
31.3.7 `assert_throws(closure)` . . . 790
31.3.8 Summary of Built-in Assertions . . . 791
31.4 The Test Standard Library Module . . . 791
31.4.1 Importing the Test Library . . . 791
31.4.2 Organizing Tests with `describe` and `it` . . . 792
31.4.3 Extended Assertions in the Test Library . . . 795
31.4.4 How the Test Library Works Internally . . . 796
31.5 Testing Strategies for Lattice Programs . . . 797
31.5.1 Unit Testing: One Function, One Test . . . 797
31.5.2 Testing Edge Cases . . . 798
31.5.3 Testing Error Paths . . . 798
31.5.4 Testing Structs and Methods . . . 799
31.5.5 Organizing a Test Suite . . . 800
31.5.6 The Test-First Workflow . . . 801
31.5.7 Testing with GC Stress Mode . . . 802
31.5.8 Disabling Assertions for Performance Testing . . . 803
Exercises . . . 804
What's Next . . . 804

32 Debugging 805
32.1 The `-debug` Flag and the CLI Debugger . . . 806
32.1.1 The Help Command . . . 807
32.1.2 Initial Breakpoint with `-break` . . . 807

32.1.3 Viewing Source Context . . . 808
32.1.4 Inspecting Variables . . . 809
32.1.5 Evaluating Expressions . . . 810
32.1.6 The Call Stack . . . 813
32.2 Breakpoints: Line, Function, Conditional . . . 814
32.2.1 Line Breakpoints . . . 814
32.2.2 Function Breakpoints . . . 814
32.2.3 Conditional Breakpoints . . . 815
32.2.4 Managing Breakpoints . . . 816
32.3 Step-Into, Step-Over, Step-Out . . . 817
32.3.1 Step-Into (s) . . . 817
32.3.2 Step-Over (n) . . . 818
32.3.3 Step-Out (o) . . . 819
32.3.4 Continue (c) . . . 820
32.4 Watch Expressions and `breakpoint()` . . . 821
32.4.1 Watch Expressions . . . 821
32.4.2 The `breakpoint()` Built-in . . . 822
32.5 DAP and VS Code Integration . . . 826
32.5.1 What Is DAP? . . . 826
32.5.2 Starting DAP Mode . . . 826
32.5.3 DAP Capabilities . . . 827
32.5.4 What the Adapter Sends . . . 827
32.5.5 Connecting an Editor . . . 829
32.6 Value History: `track()`, `history()`, `rewind()` . . . 830
32.6.1 `track(variable)` — Start Recording . . . 831
32.6.2 `history(variable)` — View the Timeline . . . 831
32.6.3 `rewind(variable, n)` — Look Back in Time . . . 833
32.6.4 Practical Example: Debugging a Running Total . . . 834
32.6.5 Combining Tracking with the Debugger . . . 836
32.7 Diagnostics Without a Prompt: `-stats` and the GC Flags . . . 837
32.7.1 `-stats` . . . 838
32.7.2 The GC Flags . . . 839
Exercises . . . 840
What's Next . . . 841

X Under the Hood **843**

33 The Three Backends **845**
33.1 Tree-Walk Interpreter . . . 845

33.1.1 How It Works . . . 845
33.1.2 Environment and Scoping . . . 846
33.1.3 When to Use It . . . 847
33.2 Stack-Based Bytecode VM . . . 848
33.2.1 Compilation . . . 848
33.2.2 Opcodes . . . 849
33.2.3 The Value Stack . . . 850
33.2.4 Call Frames . . . 850
33.2.5 The Dispatch Loop . . . 851
33.2.6 Exception Handling and Defer . . . 851
33.2.7 Upvalues and Closures . . . 852
33.3 Register-Based VM . . . 853
33.3.1 Instruction Encoding . . . 853
33.3.2 Register Windows . . . 854
33.3.3 The Register Compiler . . . 855
33.3.4 Inline Caches . . . 855
33.3.5 RegChunk: The Register VM's Code Container . . . 856
33.4 Choosing a Backend . . . 857
33.4.1 Command-Line Flags . . . 857
33.4.2 Feature Support . . . 857
33.4.3 Divergences That Remain in v0.6.4 . . . 858
33.4.4 The Execution Pipeline . . . 860
33.5 Performance Characteristics . . . 860
33.5.1 Where the Time Goes . . . 860
33.5.2 Microbenchmark: Fibonacci . . . 861
33.5.3 Startup Time . . . 862
33.5.4 Memory Usage . . . 862
33.5.5 Which One Should You Choose? . . . 862
33.6 Exercises . . . 862

34 Bytecode Serialization **865**
34.1 The .latc and .rlat File Formats . . . 865
34.1.1 Header . . . 866
34.1.2 Three Format Generations . . . 867
34.1.3 Chunk Serialization (Stack VM) . . . 870
34.1.4 RegChunk Serialization (Register VM) . . . 872
34.1.5 Endianness and Portability . . . 873
34.2 Wide Constant Opcodes . . . 874
34.3 Pre-Compiling for Faster Startup . . . 875
34.3.1 The Compile Command . . . 875

34.3.2 Running Pre-Compiled Bytecode . . . 876
34.3.3 When to Pre-Compile . . . 878
34.3.4 The Thin Runtime . . . 879
34.4 Load-Time Verification . . . 879
34.5 The Self-Hosted Compiler . . . 881
34.5.1 A Compiler Written in Lattice . . . 881
34.5.2 Full Parity . . . 882
34.5.3 Structure of the Self-Hosted Compiler . . . 882
34.5.4 How It Produces .latc Files . . . 885
34.6 Bootstrapping . . . 886
34.6.1 The Chicken-and-Egg Problem . . . 886
34.6.2 The Fixed Point . . . 887
34.6.3 Why Self-Hosting Matters . . . 888
34.6.4 The Bootstrap in Practice . . . 889

35 Native Extensions **891**
35.1 The Extension API . . . 891
35.1.1 The Contract . . . 891
35.1.2 Loading an Extension from Lattice . . . 892
35.1.3 The LatExtFn Signature . . . 893
35.2 Value Constructors and Accessors . . . 893
35.2.1 Constructors . . . 893
35.2.2 Type Queries . . . 894
35.2.3 Accessors . . . 894
35.2.4 Cleanup . . . 895
35.3 Building a .dylib/.so Extension . . . 895
35.3.1 Step 1: Write the C Code . . . 895
35.3.2 Step 2: Write a Makefile . . . 896
35.3.3 Step 3: Build and Install . . . 897
35.3.4 Step 4: Use It from Lattice . . . 898
35.4 Available Extensions . . . 898
35.4.1 ffi — Foreign Function Interface . . . 899
35.4.2 sqlite — SQLite Database . . . 900
35.4.3 pg — PostgreSQL . . . 900
35.4.4 redis — Redis Client . . . 901
35.4.5 image — Image Metadata and Operations . . . 902
35.4.6 websocket — WebSocket Client and Server . . . 903
35.5 Extension Search Paths . . . 904
35.5.1 Security Considerations . . . 906
35.5.2 WASM Limitation . . . 906

36 Metaprogramming and Reflection **909**
36.1 Runtime Code Introspection and Dynamic Features . . . 909
36.1.1 Type Queries . . . 909
36.1.2 Phase Queries . . . 911
36.1.3 String Representations . . . 912
36.1.4 Dynamic Evaluation with lat_eval() . . . 913
36.1.5 Completeness Checking with is_complete() . . . 913
36.2 tokenize() — Inspecting Tokens . . . 914
36.2.1 Use Cases for tokenize() . . . 915
36.2.2 Token Types . . . 917
36.3 Struct Reflection Functions . . . 917
36.3.1 struct_name() . . . 918
36.3.2 struct_fields() . . . 919
36.3.3 struct_to_map() . . . 919
36.3.4 struct_from_map() . . . 920
36.3.5 Combining Reflection for Generic Programming . . . 921
36.4 format() — String Building . . . 923
36.4.1 Basic Usage . . . 923
36.4.2 Escaping Braces . . . 924
36.4.3 Error Handling . . . 924
36.4.4 format() vs. String Interpolation . . . 924
36.5 The Power and Danger of Runtime Features . . . 925
36.5.1 The Power . . . 925
36.5.2 The Dangers . . . 927
36.5.3 Guidelines for Responsible Metaprogramming . . . 928
36.6 Exercises . . . 929

XI Real-World Lattice **931**

37 Building a CLI Tool **933**
37.1 The `cli` Library for Argument Parsing . . . 933
37.1.1 One Space Before You Begin . . . 934
37.1.2 Creating a Parser . . . 935
37.1.3 Flags, Options, and Positional Arguments . . . 935
37.1.4 Parsing and the Result Map . . . 936
37.1.5 Automatic Help and Error Handling . . . 937
37.1.6 Combined Short Flags . . . 939
37.2 The `dotenv` Library for Configuration . . . 939
37.2.1 The Basics: `load` and `env` . . . 939

37.2.2 Loading Specific Files . . . 940
37.2.3 Advanced Loading with Options . . . 941
37.2.4 Parsing Rules . . . 941
37.3 The `log` Library for Structured Logging . . . 943
37.3.1 One Backend Only, For Now . . . 943
37.3.2 Quick Start: Module-Level Functions . . . 945
37.3.3 Log Levels . . . 946
37.3.4 Creating a Custom Logger . . . 946
37.3.5 Structured Context . . . 947
37.4 Putting It All Together: A Complete CLI Application . . . 949
37.4.1 Project Layout . . . 949
37.4.2 The Complete Source . . . 952
37.4.3 Running the Tool . . . 953
37.4.4 Design Walkthrough . . . 954
37.5 Driving Other Programs with `exec_argv` . . . 955
37.6 Shipping the Tool . . . 957

38 Building a Web Service **961**
38.1 The `http_server` Library . . . 961
38.1.1 Hello, Web . . . 961
38.1.2 Response Helpers . . . 963
38.1.3 Route Registration . . . 964
38.1.4 Query Parameters . . . 965
38.1.5 Middleware . . . 966
38.1.6 Cookies . . . 969
38.2 The `template` Library for HTML Templating . . . 971
38.2.1 Variables and Escaping . . . 971
38.2.2 Filters . . . 972
38.2.3 Conditionals . . . 973
38.2.4 Loops . . . 975
38.2.5 Includes and Template Inheritance . . . 975
38.3 The `orm` Library for SQLite Persistence . . . 977
38.3.1 Backend and Extension Prerequisites . . . 977
38.3.2 Connecting and Defining Models . . . 978
38.3.3 CRUD Operations . . . 979
38.4 JSON APIs with `json_parse` and `json_stringify` . . . 981
38.4.1 Parsing JSON . . . 981
38.4.2 Serializing to JSON . . . 982
38.4.3 JSON in HTTP Handlers . . . 983
38.5 A Complete Web Application from Scratch . . . 984

38.5.1 Architecture . . . 985
38.5.2 Templates . . . 985
38.5.3 The Application Source . . . 989
38.5.4 Testing It Out . . . 990
38.5.5 Design Walkthrough . . . 992

A Language Grammar Reference **995**
A.1 Notation . . . 995
A.2 Program Structure . . . 996
A.3 Declarations . . . 996
A.3.1 Function contracts . . . 997
A.4 Type Expressions . . . 999
A.4.1 Function type annotations . . . 999
A.5 Statements . . . 1000
A.5.1 Import forms . . . 1000
A.6 Expressions . . . 1001
A.6.1 Postfix Expressions . . . 1002
A.6.2 Primary Expressions . . . 1003
A.6.3 Control-Flow Expressions . . . 1004
A.6.4 Phase Expressions . . . 1005
A.6.5 The freeze modifiers . . . 1005
A.6.6 Other Expressions . . . 1006
A.7 Patterns . . . 1006
A.7.1 Enum variant patterns . . . 1007
A.8 Lexical Grammar . . . 1008
A.8.1 Keywords . . . 1008
A.8.2 Literals . . . 1009
A.8.3 Operators and Delimiters . . . 1010
A.8.4 Compound Assignment Operators . . . 1010
A.8.5 Comments . . . 1010
A.9 Parser Limits . . . 1011

B Opcode Reference **1013**
B.1 Stack VM Opcodes . . . 1014
B.1.1 Stack Manipulation . . . 1014
B.1.2 Arithmetic and Logic . . . 1015
B.1.3 Bitwise Operations . . . 1015
B.1.4 Comparison . . . 1016
B.1.5 Variables and Upvalues . . . 1016
B.1.6 Control Flow . . . 1017

B.1.7 Functions and Closures . . . 1017
B.1.8 Iterators . . . 1017
B.1.9 Data Structures . . . 1017
B.1.10 Method Invocation . . . 1018
B.1.11 Exception Handling and Defer . . . 1019
B.1.12 Phase System . . . 1020
B.1.13 Pattern Matching . . . 1021
B.1.14 Builtins, I/O, and Modules . . . 1021
B.1.15 Concurrency . . . 1021
B.1.16 Optimization Fast-Path . . . 1022
B.1.17 Type Checking, Arena, and Halt . . . 1022
B.2 Register VM Opcodes . . . 1023
B.2.1 Data Movement . . . 1023
B.2.2 Arithmetic . . . 1024
B.2.3 Comparison and Logic . . . 1024
B.2.4 Bitwise Operations . . . 1024
B.2.5 Control Flow . . . 1024
B.2.6 Variables, Fields, and Upvalues . . . 1024
B.2.7 Functions . . . 1024
B.2.8 Data Structures . . . 1024
B.2.9 Builtins and Method Invocation . . . 1024
B.2.10 Iterators, Phase, Exceptions, and Defer . . . 1024
B.2.11 Advanced Phase, Concurrency, and Modules . . . 1024
B.2.12 Pattern Matching . . . 1025
B.2.13 Optimization Fast-Path and Halt . . . 1025
B.3 Bytecode Files . . . 1025
B.3.1 Sub-chunk constants . . . 1026
B.3.2 Load-time verification . . . 1026

C Phase Quick Reference **1031**
C.1 Phase Tags at a Glance . . . 1031
C.2 Binding Keywords . . . 1032
C.3 Phase Transition Operations . . . 1032
C.4 Advanced Phase Constructs . . . 1034
C.5 Type Annotations with Phases . . . 1034
C.6 Reactive Phase System . . . 1035
C.7 Phase Queries . . . 1035
C.8 Mode Comparison . . . 1035
C.9 Pattern Matching with Phases . . . 1035
C.10 Per-Field Phase Control . . . 1036

C.11 Phase Dispatch (Overloading) 1037

D Built-in Functions Reference **1039**

D.1 Global Functions 1039
D.2 Array Methods 1040
D.2.1 Non-Closure Methods 1040
D.2.2 Closure Methods 1040
D.3 String Methods 1041
D.4 Map Methods 1041
D.5 Set Methods 1041
D.6 Buffer Methods 1042
D.7 Enum Methods 1042
D.8 Universal Properties 1042

Part I

First Contact

Chapter 1

Hello, Lattice

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

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

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

1.1 What Lattice Is and Why It Exists

Lattice is an interpreted, general-purpose programming language implemented in C. It compiles your source code to bytecode and executes it on a stack-based virtual machine, with the option to run on a register-based VM or a tree-walking interpreter as well. The current release is v0.6.4, and it runs on macOS, Linux, and Windows, with an experimental WebAssembly build for the browser.

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

Listing 1.1: A first glance at Lattice syntax

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

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

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

1.1.1 The Phase System

Most languages draw a binary line: a variable is either mutable or immutable. Lattice replaces that binary with a richer model. A value starts *fluid*—you can modify it, grow it, reshape it. When you are done, you *freeze* it, and it becomes *crystal*—locked, permanent, safe to share across threads. If you need to modify it again later, you *thaw* it, and the crystal melts back into something you can reshape.

This is not merely a convention. The Lattice runtime enforces it—as of v0.5.0, on every execution backend. Try to mutate a frozen value and you will get a runtime error, complete with a hint about how to thaw it—not a silent bug discovered in production at 2 a.m.

Listing 1.2: Freezing and thawing

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

freeze(temperatures)
// temperatures.push(80.0)
//   -- error: cannot call mutating method 'push' on crystal
//      value 'temperatures' (use thaw(temperatures) to make
//      it mutable)

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

Thawing a name melts that name: after `thaw(temperatures)`, the `temperatures` binding is fluid again, holding exactly the data it held at the moment it froze. Any crystal copy already handed to somebody else stays frozen and unchanged—which is the whole point of freezing in the first place.

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

1.1.2 Familiar Syntax, Modern Features

Lattice ships with first-class closures, pattern matching with exhaustiveness checking, structs with traits and `impl` blocks, string interpolation, lazy iterators, structured concurrency with channels, and a built-in test framework. It is a language designed for working programmers who want to get things done without drowning in ceremony.

1.1.3 Batteries Included

Lattice includes over 200 built-in functions spanning file I/O, HTTP, JSON/TOML/YAML parsing, regular expressions, cryptography, networking, and more. The standard library modules can be imported with a single line. There is also a package manager for third-party libraries.

Lattice Is Young

Lattice is at version 0.6.4. The language is actively evolving. Some features described in this book may change in future releases, though the core concepts—the phase system, the type model, the concurrency story—are stable foundations that the rest of the language builds upon.

1.2 Installing Lattice

There are two ways to get Lattice onto your machine: install it with Homebrew, or build it from source with a C11 compiler and `make`. Building from source takes about a minute—part of the charm of a young language is that you *can* compile it yourself.

1.2.1 The Quick Path: Homebrew

On macOS or Linux, the fastest route is the Homebrew tap:

```
brew tap ajokela/lattice
brew install lattice
```

That gives you the `clat` binary on your `PATH`, and you can skip ahead to Section 1.2.7. If you prefer to build from source—or you are on Windows—read on.

1.2.2 Prerequisites

You need two things:

1. A **C11-compatible compiler**: GCC or Clang on Linux, Apple Clang on macOS, or MinGW on Windows.

2. **libedit**: a lightweight readline library used for the REPL's line editing and tab completion. It ships with macOS. On Linux, install the development package.

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

1.2.3 macOS

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

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

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

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

1.2.4 Linux

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

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

On Fedora or RHEL:

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

1.2.5 Windows

Lattice includes a Win32 compatibility layer (`src/win32_compat.c`) that provides POSIX-like functions, and native Windows builds have been exercised in continuous integration since v0.3.29. Using MinGW or WSL:

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

WSL Is the Smoothest Path

If you are on Windows, the Windows Subsystem for Linux gives you the best experience. Lattice was primarily developed on macOS and Linux, and WSL brings you into that environment without dual-booting. Native Windows builds work too, with two caveats: TLS support arrived in v0.4.1 (earlier Windows builds could not make `https` connections), and the POSIX regex built-ins are unavailable on Windows.

1.2.6 WebAssembly

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

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

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

1.2.7 Verifying the Installation

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

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

You should see:

```
Lattice v0.6.4
```

(`-V` is the short form.) If you want the full help screen with all available subcommands and flags:

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

The help screen lists eight subcommands: `compile`, `test`, `fmt`, `doc`, `init`, `install`, `add`, and `remove`. Under them come the runtime options—`--tree-walk` and `--regvm` to select alternate execution backends, `--stats` to print a memory-statistics block after a run (it works on all three backends), `--debug` and `--dap` for the debugger, and a family of garbage-collector flags (`--gc`, `--no-gc`, `--gc-stress`, `--gc-incremental`). Every subcommand has its own help too: try `./clat fmt --help`. We will meet these tools properly in Part IX.

1.3 Your First Program: Hello, World!

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

Listing 1.3: hello.lat — your first Lattice program

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

Run it:

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

Output:

```
Hello, World!
```

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

1.3.1 A Slightly More Interesting Program

Let's make it personal:

Listing 1.4: greeting.lat — string interpolation

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

Output:

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

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

1.3.2 Functions

Here is a program with a function:

Listing 1.5: A function with a type annotation

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

let boiling = fahrenheit_to_celsius(212.0)
print("Water boils at ${boiling} Celsius")
// Water boils at 100 Celsius
```

A few things to notice:

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

- The `-> Float` after the parameter list declares the return type.
- The output says `100`, not `100.0`. When a `Float` holds a whole number, Lattice prints it without the decimal point. The value is still a `Float`—only the display is compact.

1.3.3 Running the Examples Directory

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

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

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

Listing 1.6: A taste of the Fibonacci example

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

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

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

1.4 Running Files vs. the REPL

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

1.4.1 Running from a File

You've already seen this:

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

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

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

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

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

The `.latc` file contains serialized bytecode. Running a pre-compiled file skips the lexing, parsing, and compilation steps, which can speed up startup for larger programs. Any file ending in `.latc` (or `.rlat`, the register VM's format, produced by `./clat compile --regvm program.lat -o program.rlat`) is executed as pre-compiled bytecode automatically. There is even a separate build target—`make runtime`—that produces `clat-run`, a thin runtime which executes both bytecode formats without carrying a lexer, parser, compiler, formatter, language server, or debugger. On macOS it comes out at roughly half the size of the full `clat` binary.

Three Execution Backends

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

1.4.2 The Interactive REPL

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

```
./clat
```

You'll see a welcome banner:

```
Lattice v0.6.4 — crystallization-based programming language
Copyright (c) 2026 Alex Jokela. BSD 3-Clause License.
Type expressions to evaluate. Ctrl-D to exit.
```

Now you can type expressions and see results immediately:

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

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

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

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

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

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

1.4.3 Passing Arguments to Scripts

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

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

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

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

Output:

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

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

1.5 A Taste of What Makes Lattice Different

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

1.5.1 The Phase System in Action

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

Listing 1.8: Building and freezing a configuration

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

// Lock it down
freeze(config)

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

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

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

Listing 1.9: Forge: build fluid, end crystal

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

1.5.2 Pattern Matching

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

Listing 1.10: Pattern matching

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

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

1.5.3 Structured Concurrency

Lattice uses `scope` and `spawn` for structured concurrency. Every spawned task must complete before its enclosing scope exits, and the phase system decides what may travel between threads—frozen values always qualify, which is why we freeze these strings before sending them:

Listing 1.11: Structured concurrency with channels

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

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

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

Notice: values sent on a channel must follow the send eligibility rules: crystal (frozen), unphased, or scalars are allowed; fluid containers, sublimated values, and values containing closures, Refs, or iterators are rejected (see Section 20.1.2). This is the phase system doing its job—guaranteeing that shared data is safe.

1.5.4 Closures and Functional Style

Closures are first-class citizens in Lattice:

Listing 1.12: Closures and higher-order functions

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

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

print(total)  // 60
```

And you can chain the same transforms through a lazy iterator:

Listing 1.13: Chaining transforms with iterators

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

print(total)  // 60
```

1.5.5 String Interpolation Everywhere

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

Listing 1.14: String varieties

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

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

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

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

1.5.6 One More Thing: Strict Mode

At the top of any file, you can write `#mode strict` to opt into stricter phase discipline. In strict mode, a phase checker runs before execution and demands that every binding declare its intentions—`let` is not allowed:

Listing 1.15: Strict mode

```
#mode strict

let x = 5
// phase error: strict mode: use 'flux' or 'fix'
// instead of 'let' for binding 'x'
```

Strict mode also gives `freeze()` *consuming* semantics: freezing a variable removes the original binding, so the fluid name cannot be touched again after its value crystallizes. (As of v0.6.4, this particular rule is enforced at runtime by the tree-walking interpreter—accessing the consumed name reports `undefined variable`—while the bytecode VMs do not yet enforce it.)

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

1.6 Exercises

1. **Hello, You.** Write a program that asks for the user's name using `input("What is your name? ")` and prints a personalized greeting using string interpolation.

2. **Temperature Table.** Write a program that prints a conversion table from Fahrenheit to Celsius for every 10 degrees from 0 to 212. Use a `for` loop and the range operator. Hint: `for f in range(0, 220, 10) { ... }`

3. **Explore the REPL.** Launch the REPL (`./clat`) and experiment. Define a function, call it with different arguments, and try to break it by passing the wrong types. What error messages do you get?

4. **Freeze Tag.** In the REPL, create an array with `flux items = [10, 20, 30]`. Push a new value onto it. Then freeze it with `freeze(items)` and try to push again. Read the error message—it will become familiar.

5. **Read an Example.** Read the source of `examples/phase_demo.lat` and run it. Before running, predict what each `print` statement will output. Were you right?

What's Next

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

Chapter 2

The REPL Is Your Laboratory

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

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

2.1 Launching the REPL

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

```
./clat
```

You'll see the welcome banner:

```
Lattice v0.6.4 — crystallization-based programming language
Copyright (c) 2026 Alex Jokela. BSD 3-Clause License.
Type expressions to evaluate. Ctrl-D to exit.
```

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

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

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

2.1.1 Persistent State Across Lines

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

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

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

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

REPL Persistence

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

2.1.2 Multi-line Input

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

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

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

This works for any kind of nesting: function bodies, `if`/`else` blocks, `match` expressions, struct definitions, and more. An unterminated string holds the prompt open the same way, which means you can type a multi-line string straight into the REPL and close the quote whenever you are ready. The accumulation buffer is bounded: as of v0.6.4, input past 64 KB is rejected with `error: REPL input too long (max 65535 bytes); resetting buffer` rather than growing without limit.

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

Escaping a Multi-line Mistake

If you start typing a multi-line expression and realize you've made a mistake, close all open delimiters to force evaluation. That will likely produce a parse error, but the error is harmless—the buffer clears, and the next `lattice>` prompt starts fresh. Do not reach for `Ctrl-C`: the REPL installs no interrupt handler, so `Ctrl-C` terminates the whole session, definitions and all.

2.2 Tab Completion and Discoverability

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

2.2.1 Keyword and Built-in Completion

Start typing a keyword or built-in function name and press Tab. With a single match, the line completes in place:

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

If there are multiple matches, the first Tab fills in the longest common prefix, and a second Tab lists all the options:

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

The completion engine knows about all of Lattice's keywords (`fn`, `let`, `flux`, `fix`, `match`, `struct`, `enum`, `trait`, `impl`, and more) and over 190 built-in functions. This makes the REPL a discovery tool: if you vaguely remember that there's a function related to files, type `file` and press Tab twice:

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

(The first Tab extends the line to `file_`, the shared prefix; the second shows both candidates.)

2.2.2 Constructor Completion

Constructor names like `Map::new` are completion entries in their own right, so a type name followed by `::` completes like any other word:

```
lattice> Channel::<Tab>
lattice> Channel::new
```

When a type has more than one constructor, Tab-Tab lists them:

```
lattice> Buffer::<Tab><Tab>
Buffer::from        Buffer::from_string Buffer::new
```

2.2.3 What About Method Names?

`src/completion.c` also carries a table of a hundred-odd method names, intended to complete after a dot (`scores.fi<Tab>` suggesting `filter`, `find`, and friends). In the current release, though, the line editor treats a dotted expression such as `scores.fi` as a single word, so the method table is never consulted—pressing `Tab` after a dot does nothing (verified against v0.6.4). Method discovery is one place where the REPL still makes you reach for this book.

How Completion Works Internally

The tab completion system is implemented in `src/completion.c`. It registers a custom `rl_attempted_completion_function` with the readline library and defines three modes: `COMPLETE_GLOBAL` (keywords, built-in functions, and constructor names—235 entries in all), `COMPLETE_METHOD` (after a dot), and `COMPLETE_SCOPE` (after `::`). In practice the line editor's word-splitting delivers whole tokens like `Map::` or `scores.fi` to the completer, so the global list does all the work—which is why `Map::new` completes (it is a global entry) while method names after a dot do not. Filename completion is explicitly disabled—the REPL suggests Lattice symbols, not files.

2.3 Defining Functions, Structs, and Enums Interactively

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

2.3.1 Functions

Define a function and call it immediately:

```
lattice> fn celsius_to_fahrenheit(c: Float) -> Float {
    ...>     c * 9.0 / 5.0 + 32.0
    ...> }
lattice> celsius_to_fahrenheit(100.0)
=> 212
lattice> celsius_to_fahrenheit(0.0)
=> 32
```

Notice that the results display as `212` and `32`, not `212.0` and `32.0`. Floats that hold whole numbers print without the decimal point; `typeof(celsius_to_fahrenheit(100.0))` confirms the value is still a `"Float"`.

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

```
lattice> let temps_c = [0.0, 20.0, 37.0, 100.0]
lattice> temps_c.map(|c| celsius_to_fahrenheit(c))
=> [32, 68, 98.6, 212]
```

2.3.2 Structs

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

```
lattice> struct Point {
    ...>     x: Float,
    ...>     y: Float
    ...> }
lattice> let origin = Point { x: 0.0, y: 0.0 }
lattice> let target = Point { x: 3.0, y: 4.0 }
lattice> target.x
=> 3
```

Add a trait and implement it:

```
lattice> trait Measurable {
    ...>     fn distance(self: Any) -> Float
    ...> }
lattice> impl Measurable for Point {
    ...>     fn distance(self: Any) -> Float {
    ...>         sqrt(self.x * self.x + self.y * self.y)
    ...>     }
    ...> }
lattice> target.distance()
=> 5
```

2.3.3 Enums

Enums work the same way:

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

```
lattice> enum Shape {
    ...>     Circle(Float),
    ...>     Rectangle(Float, Float)
    ...> }
lattice> let s = Shape::Rectangle(4.0, 5.0)
lattice> s.payload()
=> [4, 5]
```

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

2.4 The => Prefix: How the REPL Talks Back

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

2.4.1 The Display Rule

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

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

Statements that produce `unit` are silently suppressed:

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

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

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

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

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

2.4.2 The `repr` Format

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

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

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

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

This distinction between `repr` (for programmers, with type information) and `print` (for users, raw output) is consistent throughout Lattice. A struct can carry its own `repr` closure field, and the `repr()` built-in calls it on every backend:

```
lattice> struct Temp { c: Float, repr: Any }
lattice> let t = Temp { c: 21.5, repr: |self| { "Temp(${self.c}C)" } }
lattice> repr(t)
=> "Temp(21.5C)"
lattice> t
=> Temp { c: 21.5, repr: <closure|> }
```

The automatic `=>` display is the one place that does not consult it: the bytecode REPLs fall back to the structural form on that last line. The tree-walking REPL does call the custom closure—one of the few places where the three prompts disagree about what to show you.

2.4.3 Distinguishing `print` from =>

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

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

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

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

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

Unit and Nil

`unit` is the return type of statements and functions with no meaningful return value—like variable assignments or `print()`. `nil` is an explicit null value that means "no value here." Both are suppressed in REPL output, but they are different types. `typeof(nil)` returns `"Nil"`. There is no `unit` literal to hand to `typeof`, so to see the other one you capture a statement's result—`let u = print("")` followed by `typeof(u)` answers `"Unit"`. We will cover both in Chapter 3.

2.5 Tips and Tricks for Exploratory Programming

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

2.5.1 Use `typeof()` to Inspect Values

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

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

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

2.5.2 Use `phase_of()` to Inspect Phase

Check whether a value is fluid or crystal:

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

(`freeze()` returns the value it froze, so the REPL echoes the array.)

2.5.3 Build Data Structures Incrementally

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

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

You can see the state of your data at each step, catch mistakes early, and iterate quickly. Two display quirks worth knowing: maps list their entries in internal hash order, not insertion order (`"oranges"` jumped ahead of `"bananas"` here), and strings inside a displayed array appear without quotes.

2.5.4 Test Functions with Edge Cases

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

```
lattice> fn safe_divide(a: Float, b: Float) -> Float {
    ...>     if b == 0.0 { return 0.0 }
    ...>     a / b
    ...> }
lattice> safe_divide(10.0, 3.0)
=> 3.33333
lattice> safe_divide(10.0, 0.0)
=> 0
lattice> safe_divide(-7.5, 2.5)
=> -3
```

The REPL gives you a tight feedback loop for testing behavior before writing formal tests. (Float display rounds to six significant digits—the stored value keeps full precision.)

2.5.5 Use History to Save Typing

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

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

2.5.6 Explore the Standard Library

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

```
lattice> version()
=> "0.6.4"
```

Curious about the current platform?

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

Want to see what time it is?

```
lattice> time()
=> 1785901530975
lattice> time_format(time(), "%Y-%m-%d %H:%M:%S")
=> "2026-08-04 22:45:30"
```

Need to know your working directory?

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

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

2.5.7 Error Recovery

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

```
lattice> let x = 42
lattice> x.push(3)
error: type 'Int' has no method 'push'
lattice> x
=> 42
```

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

Note the bare `error:` prefix. The REPL announces failures that way on all three backends—never `vm error:`, never `regvm error:`—so a transcript in this book that shows a prefixed message is a *file* run, not a prompt. Run the same mistake from a source file and the prefix appears: `vm error:` from the stack VM, `regvm error:` from the register VM, and a bare `error:` from `--tree-walk`, which happens to match what its REPL prints. The wording after the prefix can differ between backends too, and this particular refusal comes out three different ways: `type 'Int' has no method 'push'` on the default stack VM, `.push() is not defined on non-array` under `--tree-walk`, and `no method 'push' on Int` under `--regvm`. Whenever this book quotes a missing-method message, it names the backend that said it. The outcome, though, is the same everywhere: a message, an intact session, and a fresh prompt.

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

2.5.8 The Debugging REPL

For an even more powerful interactive experience, you can embed `breakpoint()` calls in your source files. When the interpreter reaches one, it drops into a small interactive prompt where you can list the current frame's locals, print any one of them, walk the call stack, and evaluate expressions.

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

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

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

Running that file gives us a look inside the frame:

```
-- breakpoint() hit at line 4 --
Commands: locals, bt, p <var>, <expr>, cont

(breakpoint) locals
  items = [widget, gadget]
  total = 150
  discount = 0.1
  final_price = 135
(breakpoint) p final_price
final_price = 135
(breakpoint) bt
Backtrace:
  #0 [line 8] in <script>
  #1 [line 4] in process_order  <-- current
(breakpoint) cont
Final price: 135
```

Use `locals` and `p`, Not Bare Names

Expressions you type at the `(breakpoint)` prompt are compiled fresh, so under the bytecode VM they cannot see the frame's locals: typing `final_price` on its own reports `error: undefined variable 'final_price'` and ends the run. `p final_price` reads the frame directly and always works. (The tree-walking interpreter does resolve bare locals, and the register VM declines the invitation altogether with `breakpoint(): not supported under the register VM backend.`)

We'll cover the full debugger—`--debug`, breakpoints by line and by function name, watches, and the Debug Adapter Protocol server behind `--dap`—in Chapter 32 (*Debugging*).

2.5.9 Alternative REPL Backends

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

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

All three keep the same persistent-session model and the same `=>` display rule—only the machine underneath changes. The banner tells you where you are: the register VM appends `(regvm)` to its first line, and the tree-walking REPL prints the banner unchanged. The tree-walking REPL earns its keep when you want a second opinion: it resolves bare locals at a `breakpoint()` prompt and calls custom `repr` closures, as we saw above. The register VM REPL matches the default on the surface, minus `breakpoint()`. In normal use, the default bytecode VM REPL is the right choice.

The Self-Hosted REPL

Lattice also includes a self-hosted REPL written in Lattice itself (`repl.lat`): forty-eight lines built on `input`, `is_complete`, and `lat_eval`, with the evaluation wrapped in `try`/`catch`. Run it with `./clat repl.lat`. Two v0.6.4 refinements are what make that loop trustworthy: `input()` returns `nil` at end of input on every backend, so `Ctrl-D` ends the loop cleanly, and `lat_eval()` now raises runtime failures as catchable errors instead of quietly handing back nothing—which is why a line like `1 / 0` prints `error: division by zero` and the session carries on. It has no tab completion or line editing, and it echoes results through `print()` rather than the `=>` prefix. The C-based REPL described in this chapter is what you should use day-to-day.

A REPL With a Domain

That self-hosted loop is less a curiosity than a template. `examples/ballistics_lab/` in the Lattice repository builds a full stateful laboratory on the same three built-ins: SI-backed units, immutable domain values, an external solver invoked through `exec_argv`, DOPE and wind-card tables, and JSON/CSV export—all in ordinary Lattice, with no additions to the runtime. Start it from the repository root:

```
./clat examples/ballistics_lab/ballistics-repl.lat
Ballistics laboratory — Lattice v0.6.4
Engine: ballistics
Type :help for commands or enter Lattice. Ctrl-D to exit.
```

Colon commands (`:show`, `:solve`, `:dope`, `:wind-card`, `:save "run.json"`) drive the session; every other line is evaluated as Lattice against that same persistent state. It is worth reading when you want to see what a domain-specific REPL looks like once the language itself stops being the subject.

2.5.10 A Complete Exploratory Session

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

```
lattice> let scores = [85, 92, 78, 95, 88, 72, 91]
lattice> scores.len()
=> 7
lattice> scores.sort()
=> [72, 78, 85, 88, 91, 92, 95]
lattice> scores.min()
=> 72
lattice> scores.max()
=> 95
lattice> scores.sum()
=> 601
lattice> scores.filter(|s| s >= 90)
=> [92, 95, 91]
lattice> scores.filter(|s| s >= 90).len()
=> 3
lattice> let avg = to_float(scores.sum()) / to_float(scores.len())
lattice> avg
=> 85.8571
lattice> scores.map(|s| if s >= 90 { "A" } else { "B" })
=> [B, A, B, A, B, B, A]
lattice> scores.enumerate()
=> [[0, 85], [1, 92], [2, 78], [3, 95], [4, 88], [5, 72], [6, 91]]
```

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

2.6 Exercises

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

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

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

4. **Tab Completion Safari.** In the REPL, type `str` and press `Tab` twice. What completions appear, and which of them are keywords rather than functions? Now try `Buffer::` with two `Tab`s, and `Map::` with one. Finally type `"hello".to` and press `Tab`: what happens, and why?

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

What's Next

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

Chapter 3

Values, Types, and the Shape of Things

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

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

3.1 Integers

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

Listing 3.1: Integer basics

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

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

Integer Range

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

3.2 Floats

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

Listing 3.2: Float basics

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

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

Integer Division vs. Float Division

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

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

Lattice provides two functions for inspecting special float values:

Listing 3.3: Special float values

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

You can convert between integers and floats explicitly:

Listing 3.4: Numeric conversions

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

One display quirk worth knowing before we go further: `print` formats floats compactly, showing up to six significant digits, and whole-number floats drop their decimal point entirely:

Listing 3.5: How floats print

```
print(-40.0)        // -40
print(3.5)          // 3.5
print(3.14159265)   // 3.14159
print(1234567.0)    // 1.23457e+06
```

The values are still floats—`typeof(-40.0)` is `Float`—only the display is compact. Keep this in mind whenever an example in this book prints `0` where you expected `0.0`.

3.3 Booleans

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

Listing 3.6: Boolean basics

```
let is_valid = true
let has_errors = false

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

3.3.1 Truthiness

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

- `false` is falsy.
- `nil` is falsy.
- `0` and `0.0` (numeric zero) are falsy.
- `""` (empty string) is falsy.
- An empty `Map` and an empty `Set` are falsy.
- Everything else is truthy—including empty arrays.

Listing 3.7: Truthiness in action

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

Empty Arrays Are Truthy; Empty Maps Are Not

Unlike Python, an empty array is *truthy* in Lattice—but an empty map or set is *falsy*. This asymmetry catches almost everyone at least once. When you want to know whether a collection is empty, don't lean on truthiness: use `.len() == 0` (or `.is_empty()` for strings) and say what you mean.

3.4 Strings

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

3.4.1 Double-Quoted Strings

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

Listing 3.8: Double-quoted strings

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

The full set of escape sequences is:

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

Table 3.1: String escape sequences

There Is No Null-Byte Escape

You may notice `\0` missing from the table. Writing `\0` (or `\x00`) in a string literal does not embed a null byte—it truncates the literal at that point, so `"A\0B".len()` is `1` and `"\0".len()` is `0`. Null bytes enter strings through other doors: `Buffer` round-trips, `json_parse`, process output, sockets. Once inside, they are ordinary data—as of v0.6.4, strings are binary-safe, and `len`, `split`, indexing, equality, and friends all honor the true byte length. We'll return to binary-safe strings in Chapter 8.

3.4.2 String Interpolation

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

Listing 3.9: String interpolation

```
let language = "Lattice"
let major = 0
let minor = 6

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

print("Version ${major}.${minor}")
// Version 0.6

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

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

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

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

Listing 3.10: Escaping interpolation

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

3.4.3 Single-Quoted Strings

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

Listing 3.11: Single-quoted strings: no interpolation

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

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

Listing 3.12: Single-quoted strings for templates

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

When to Use Single vs. Double Quotes

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

3.4.4 Triple-Quoted Strings

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

Listing 3.13: Triple-quoted strings

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

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

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

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

Triple-quoted strings support interpolation:

Listing 3.14: Triple-quoted strings with interpolation

```
let name = "Lattice"
let version = "0.6.4"
let banner = """
    ============================
     ${name} v${version}
     A crystallization language
    ============================
    """
print(banner)
```

3.4.5 Common String Methods

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

Listing 3.15: String methods

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

print(message.trim())            // Hello, Lattice!
print(message.len())             // 19
print(message.contains("Lat"))   // true
print("shout".to_upper())        // SHOUT
print("Hello, Lattice!".replace("Lattice", "World"))  // Hello, World!
print("one,two,three".split(","))  // [one, two, three]

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

Four predicates round out the searching-and-classifying toolkit:

Listing 3.16: String predicates

```
print("hello world".last_index_of("o"))  // 7
print("hello".is_alpha())                // true
print("12345".is_digit())                // true
print("abc123".is_alphanumeric())        // true
print("abc 123".is_alphanumeric())       // false (space is neither)
```

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

3.5 Nil and Unit

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

3.5.1 Nil

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

Listing 3.17: Nil basics

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

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

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

Listing 3.18: Nil coalescing with ??

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

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

3.5.2 Unit

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

Listing 3.19: Unit in action

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

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

Nil vs. Unit

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

3.6 Type Checking with `typeof()`

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

Listing 3.20: typeof() for every core type

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

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

Listing 3.21: Runtime type checking

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

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

For struct instances, `typeof()` returns the generic tag `Struct`—not the struct's declared name:

Listing 3.22: typeof() with structs

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

Every struct instance answers `Struct`, whatever its declaration was called. If you need to tell one struct from another at runtime, carry a discriminating field yourself—or better, reach for an enum and `match`, which we'll meet in Part IV.

Type Annotations vs. typeof()

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

3.7 Type Annotations and Function Types

Lattice's type annotations live in one place: function signatures. Every named-function parameter *requires* an annotation—leave one off and the parser stops you cold:

Listing 3.23: Parameters require annotations

```
fn area(width: Float, height: Float) -> Float {
    width * height
}

// fn area(width, height) { ... }
// error: 1:14: parse error: expected ':', got ','
```

Annotations can also describe *functions themselves*. The `Fn(...) -> T` form spells out what a callable parameter accepts and returns:

Listing 3.24: Function type annotations

```
fn apply(f: Fn(Int) -> Int, x: Int) -> Int {
    f(x)
}

print(apply(|n| { n + 1 }, 41))  // 42
```

The bare form `Fn` remains valid when the shape doesn't matter:

Listing 3.25: Bare Fn

```
fn twice(f: Fn, x: Int) -> Int {
    f(f(x))
}

print(twice(|n| { n * 3 }, 2))  // 18
```

Functions Require Annotations; Lambdas Forbid Them

The annotation rules for named functions and lambdas are exact opposites, and mixing them up is a top source of parse errors now that `Fn(...)` types exist:

- `fn` parameters *require* annotations: `fn foo(a: any)`. Omitting one is a parse error (`expected ':', got ','`).
- Lambda parameters *forbid* annotations: write `|x| { x }`, never `|x: any| { x }`—the latter fails with `parse error: expected ',', got ':'`.

While we're near the pipes: a lambda with no parameters is written `| |` (with a space between the bars). Writing `||` means logical OR.

Generic Syntax Parses—and in One Position It Bites

You may see angle brackets in the wild: `struct Box<T> { v: Int }` or `fn id<T>(x: Int) -> Int`. As of v0.6.4 there is no type substitution and no checking against `T`, so when every *annotation* names a concrete type or `any`, the parser accepts the brackets, the program runs, and the `<T>` does nothing. There is one exception, and it is the form you are most likely to reach for. Writing a bare type-parameter name as a function parameter type or return type produces a function that dies on its first call, on all three backends:

```
fn pick<T>(x: T) -> T {
    return x
}

print(pick(42))
// vm error: function 'pick' parameter 'x' expects type T, got Int
//           (did you mean 'Fn'?)
```

The runtime type checker validates parameters and return values at call time and has never heard of type parameters, so `T` is treated as a type name that nothing matches. Struct and enum fields are not checked this way, so `struct Box<T> { value: T }` stays harmless. Chapter 18 maps the whole boundary; until then, keep generic syntax to decoration and annotate with concrete types or `any`.

3.8 Number Literals in Detail

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

3.8.1 Underscore Separators

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

Listing 3.26: Underscore separators

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

print(population)   // 8100000000
print(bytes)        // 1048576
print(precise_pi)   // 3.14159 (six significant digits on display)
```

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

3.8.2 Hexadecimal Literals

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

Listing 3.27: Hexadecimal literals

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

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

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

3.8.3 No Scientific Notation

Lattice does not currently support scientific notation in source code (like `1.5e10` or `3e-4`). The lexer reads `1.5e10` as the float `1.5` followed by an identifier `e10`, and where that pair is legal syntax it fails at runtime with `undefined variable 'e10'`—so `let x = 1.5e10` compiles and then dies. On the two bytecode VMs the message arrives with a guess attached, `(did you mean 'exp'?)`, which is

the runtime reaching for the nearest builtin name and not a hint about scientific notation. Written where a comma is expected, as in `print(1.5e10)`, the two tokens never get that far: the parser stops with `1:10: parse error: expected ',', got 'identifier'`. If you need to express very large or very small numbers, use `pow()`:

Listing 3.28: Expressing large numbers

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

3.9 Operators

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

3.9.1 Arithmetic Operators

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

Table 3.2: Arithmetic operators

The `+` operator also works for string concatenation:

Listing 3.29: String concatenation with +

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

3.9.2 Comparison Operators

The ordering operators (<, >, <=, >=) work on numbers only—comparing two strings with < raises a runtime error. The default VM words it `operands must be numbers for '<'`; the other two backends

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

Table 3.3: Comparison operators

refuse as firmly, in their own phrasing. Equality (==) is more generous: it works on strings and booleans, and uses structural comparison for arrays, maps, and structs:

Listing 3.30: Structural equality

```
print([1, 2, 3] == [1, 2, 3])   // true
print([1, 2] == [1, 2, 3])      // false
print("melt" == "melt")          // true
```

Ordering Strings

Need strings in order? `.sort()` on an array of strings compares them lexicographically, even though < does not:

```
print(["banana", "apple", "cherry"].sort())
// [apple, banana, cherry]
```

3.9.3 Logical Operators

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

Table 3.4: Logical operators

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

Listing 3.31: Short-circuit evaluation

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

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

3.9.4 Bitwise Operators

Lattice provides the full set of bitwise operations on integers:

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

Table 3.5: Bitwise operators

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

Listing 3.32: Bitwise operations in practice

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

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

3.9.5 Compound Assignment

Every arithmetic and bitwise operator has a compound assignment form:

Listing 3.33: Compound assignment operators

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

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

There Is No ++ or --

Compound assignment is as far as Lattice goes—there are no increment or decrement operators. Writing `count++` stops the parser cold:

```
flux count = 1
count++
// error: 2:7: parse error: unexpected token '+' in expression
```

The double minus is sneakier, because it *does* parse—as a subtraction of a negation:

```
flux x = 10
flux y = 3
print(x--y)  // 13, because x--y means x - (-y)
print(x)     // 10, untouched
```

Reach for `+= 1` and `-= 1` and the ambiguity disappears.

Compound Assignment and Frozen Targets

Compound assignment modifies its target in place, so the target must be mutable. Writing into a frozen container's element—say `a[0] += 1` after `freeze(a)`—raises `cannot modify a frozen value` at runtime. Rebinding a plain variable is a different story: freezing protects the *value*, not the *binding*, a distinction we take up properly in Chapter 4.

3.9.6 The Nil Coalescing Operator

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

Listing 3.34: Nil coalescing

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

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

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

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

Listing 3.35: Nil coalescing with map lookups

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

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

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

3.9.7 Optional Chaining

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

Listing 3.36: Optional chaining

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

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

3.9.8 The Range Operator

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

Listing 3.37: Range operator

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

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

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

3.10 Comments

Lattice supports three styles of comments.

3.10.1 Line Comments

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

Listing 3.38: Line comments

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

3.10.2 Block Comments

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

Listing 3.39: Block comments (nestable)

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

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

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

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

3.10.3 Doc Comments

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

Listing 3.40: Doc comments

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

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

3.11 Putting It All Together

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

Listing 3.41: A temperature converter using multiple types

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

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

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

// Build a conversion table
let temps = [0.0, 32.0, 72.0, 100.0, 212.0]
for temp in temps {
    print(convert_temp(temp, "F"))
}
/*
    0 F = -17.78 C
    32 F = 0 C
    72 F = 22.22 C
    100 F = 37.78 C
    212 F = 100 C
*/
```

(Those whole-number temperatures print without their decimal points—the compact float display from Section 3.2 at work.)

This program demonstrates:

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

- **Doc comments** on the function
- **Block comments** for expected output
- The `if`/`else` expression returning a value

3.12 Exercises

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

What's Next

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

Chapter 4

Variables and the Idea of Phase

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

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

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

4.1.1 `let` — Inferred Phase

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

Listing 4.1: Declaring variables with let

```
let name = "Lattice"      // a string
let version = 64           // an integer
let pi = 3.14159           // a float
let active = true          // a boolean
let colors = ["red", "green", "blue"]  // an array
```

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

Listing 4.2: let bindings are fluid in casual mode

```
let score = 100
score = 200    // reassignment works in casual mode
print(score)   // 200

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

If you ask `phase_of()` about a fresh `let` binding, it reports `unphased`—the value has not committed to being fluid or crystal, and behaves as mutable until you say otherwise.

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

4.1.2 `flux` — Explicitly Fluid

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

Listing 4.3: Declaring mutable variables with flux

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

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

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

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

Fluid Phase

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

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

4.1.3 `fix` — Explicitly Crystal

The `fix` keyword declares a variable whose value is *crystal*—immutable, locked, permanent. The declaration freezes the value for you, so an explicit `freeze()` is optional:

Listing 4.4: Declaring immutable variables with fix

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

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

Attempting to modify a crystal value produces a runtime error—and the message even tells you the way out:

Listing 4.5: Crystal values reject modification

```
fix config = [1, 2, 3]
config.push(4)
// error: cannot call mutating method 'push' on crystal value
//        'config' (use thaw(config) to make it mutable)
```

4.1.4 Freeze Protects the Value, Not the Binding

Here is the corner of the phase system that surprises nearly everyone, so let's meet it early: a `fix` declaration freezes the *value*, but in casual mode it does *not* seal the *name*. Plain rebinding of a `fix` variable is allowed:

Listing 4.6: Rebinding a fix variable is not an error

```
fix x = 1
x = 5              // no error -- the binding is rebound
print(x)           // 5

fix a = [1]
a = [2, 2]         // the new value replaces the old one...
print(phase_of(a)) // unphased -- ...and arrives without crystal phase

fix b = [1]
// b.push(2)       -- error: in-place mutation is still blocked
```

The crystal that `freeze()` creates is a property of the frozen value itself. Whoever holds that value—any alias, any parameter, any data structure—cannot mutate it. But pointing the *name* at a different value is not a mutation of the old value; it's a new value moving in next door, and the new tenant does not inherit the old one's phase. If you want the binding itself protected, that's what `#mode strict` is for: its phase checker rejects assignments to crystal bindings before the program runs (we'll see it in Section 4.4).

Compound Assignment on Frozen Scalars: A Backend Divergence

One rough edge as of v0.6.4: `fix y = 1` followed by `y += 1` *inside a function* raises `cannot modify a frozen value` on the bytecode VMs but is silently allowed under `--tree-walk`; at top level, all backends allow it (it behaves as a rebind). Until the backends agree, don't lean on either behavior—if a scalar needs updating, declare it `flux`.

Crystal Phase

A value in the *crystal phase* is immutable. Its contents cannot be modified, its structure cannot change, and it can be safely shared across threads. In the chemistry metaphor, a crystal is solid, ordered, and permanent—like a diamond formed under pressure. You declare crystal variables with `fix`; in type annotations, the prefix `*` marks a parameter as requiring a crystal argument.

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

4.1.5 The Shorthand Prefixes: ~ and *

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

Listing 4.7: Phase shorthand in function parameters

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

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

Here, `~Map` means "a Map that must be fluid" and `*Map` means "a Map that must be crystal." The runtime checks these constraints on every call—hand a frozen map to `mutate()` and the call is refused before the body runs:

```
flux settings = Map::new()
settings["name"] = "beta"
freeze(settings)

mutate(settings)
// error: phase constraint violation in function 'mutate'
```

(The `--tree-walk` backend is chattier about the same refusal, naming the parameter and the phase it got.)

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

4.1.6 Comparing the Three Keywords

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

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

Table 4.1: Variable declaration keywords

4.2 Why Mutability Is a First-Class Concept

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

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

Listing 4.8: Phase is a property of the value, not the variable

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

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

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

The phase travels with the value. Pass a frozen array to a function and the parameter arrives crystal; alias it with `let` and the alias is crystal too. The crystal phase is embedded in the value, not in whatever name you happen to give it:

Listing 4.9: Phase follows the value through function calls

```
fn try_to_modify(data: any) {
    print(phase_of(data))
    let alias = data
    alias.push(99)
}

flux numbers = [1, 2, 3]
freeze(numbers)
try_to_modify(numbers)
// crystal
// error: cannot call mutating method 'push' on crystal value 'alias'
//        (use thaw(alias) to make it mutable)
```

One keyword does break the chain, and it is worth meeting deliberately rather than by accident.

`flux` Re-declaration Is an Implicit Thaw

Binding a crystal value into a *new* `flux` variable does not carry the crystal phase along. It privatizes the value and melts the copy:

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

flux working = numbers      // privatize, then melt
print(phase_of(working))    // fluid
working.push(99)

print(working)              // [1, 2, 3, 99]
print(numbers)              // [1, 2, 3]
print(phase_of(numbers))    // crystal
```

The original crystal is untouched. The compiler emits `OP_MARK_FLUID` for the `flux` declaration, and the VM unshares the value *before* flipping its phase tag (you can read the comment on that opcode in `src/stackvm.c`), so `flux working = numbers` is `thaw()` wearing a different hat. If you meant to keep the crystal, alias it with `let` instead.

That is the guarantee stated precisely: nobody can reach *through* a copy to damage the original. When a value is frozen, it is safe to share across threads, pass to unknown functions, or store in a long-lived data structure—because no holder of that value can modify it, and anyone who wants a mutable

version gets a copy of their own. The guarantee is enforced by the runtime, not by programmer discipline.

The Value vs. the Binding

The `flux`/`fix` keywords set the initial **phase of the value**—fluid or crystal—and that phase persists as the value is passed to other functions or stored in data structures. What they do *not* do, in casual mode, is seal the **binding**: as Section 4.1.4 showed, a `fix` name can still be pointed at a new value, and the newcomer arrives unphased. Only `#mode strict` turns assignment to a crystal binding into an error.

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

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

4.3.1 `freeze()` — Fluid to Crystal

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

Listing 4.10: Freezing a value

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

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

settings["debug"] = true
// error: cannot modify a frozen value
```

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

Listing 4.11: Deep freeze

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

print(phase_of(data))     // crystal
print(phase_of(data[0]))  // crystal -- the inner array froze too

// data.push([5, 6])
// error: cannot call mutating method 'push' on crystal value 'data'
//        (use thaw(data) to make it mutable)

// data[0].push(3)
// error: cannot call mutating method 'push' on a frozen value
```

Note the two shapes of that message. When the runtime can name the variable you reached through, it names it and offers `thaw()`; when the receiver is an anonymous element deep inside a structure, it falls back to the shorter "on a frozen value." Both are the same refusal.

4.3.2 `thaw()` — Crystal to Fluid

`thaw()` creates a **mutable copy** of a crystal value. The original's *data* is untouched:

Listing 4.12: Thawing a frozen value

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

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

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

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

There is a second half to the story, though, and it is the mirror image of the write-back you met in Section 4.3.1. Handing `thaw()` a bare variable name melts *that name* as well:

Listing 4.13: Thawing a name melts the name

```
fix original = freeze([10, 20, 30])
flux copy = thaw(original)

print(phase_of(original))  // fluid -- melted in place
original.push(99)
print(original)            // [10, 20, 30, 99]
print(copy)                // [10, 20, 30]  -- a separate array
```

Crystal copies you already handed out elsewhere stay frozen; it is the binding you named that goes fluid. Write `thaw(clone(original))` when you want `original` to stay hardened. Section 11.5.2 takes this apart properly.

(Recall from Section 4.2 that declaring a fresh `flux` binding from a crystal value does exactly this, without your having to write `thaw`. Writing it out is clearer about your intent.)

4.3.3 `clone()` — Independent Copy

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

Listing 4.14: Cloning preserves phase

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

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

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

Table 4.2: Phase transition functions

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

4.4 #mode casual vs. #mode strict

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

Listing 4.15: Mode directives

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

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

4.4.1 Casual Mode

In casual mode, Lattice is forgiving:

- `let` is allowed and produces an inferred-phase binding.
- `freeze()` transitions a value in place but does *not* consume the variable—you can still read from the binding.
- Phase mismatches at function call boundaries produce runtime errors, not compile-time errors.
- You can mix `let`, `flux`, and `fix` freely.

Listing 4.16: Casual mode: forgiving

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

// data.push(5)
// error: cannot call mutating method 'push' on crystal value 'data'
//        (use thaw(data) to make it mutable)
```

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

4.4.2 Strict Mode

Strict mode tightens the rules:

- `let` is **disallowed**. You must use `flux` or `fix` for every declaration, making your intent explicit.
- `freeze()` **consumes the binding**. After freezing a variable, the original name is meant to be gone—you capture the frozen result in a new binding and use that.
- The phase checker runs **before execution**, catching phase errors before a single statement runs.
- Assigning to a crystal binding is a phase error.

Listing 4.17: Strict mode: explicit and consuming

```
#mode strict

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

fix frozen = freeze(data)
// `frozen` is the name you're meant to use from here on.
print(frozen)  // [1, 2, 3, 4]
```

The consume-on-freeze idea prevents a common bug: modifying a value after you've shared a "frozen" reference to it. In casual mode, you could freeze a value, hand the frozen reference to another part of the program, and then accidentally mutate the original. Strict mode's answer is to take the original name out of scope entirely.

Consume-on-Freeze Is Not Yet Enforced on the Bytecode VMs

As of v0.6.4, the binding-consuming behavior lives only in the tree-walking interpreter (`src/eval.c`, where strict-mode `freeze` calls `env_remove`). Run the listing above under `--tree-walk` and a later `print(data)` fails with `undefined variable 'data'`; run it on the default VM or `--regvm` and `data` is still there, still readable, still holding the now-crystal array. The value is frozen on every backend—that part is uniform—but do not *rely* on the name disappearing. Write your strict-mode code as though it does, and it will be correct under all three.

Listing 4.18: Strict mode catches errors early

```
#mode strict

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

flux counter = 0
counter += 1

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

// Uncomment either line and nothing runs at all.
```

When to Use Strict Mode

Use `#mode strict` when:

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

Use casual mode (the default) when:

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

4.4.3 Inside the Phase Checker

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

1. Tracks the phase of every variable through a scope stack.
2. Verifies that `fix` bindings are never assigned to.
3. Ensures that arguments passed to phase-constrained parameters (`~Map`, `*Map`) have compatible phases.
4. Rejects `let` bindings—requiring explicit `flux` or `fix`.

5. Inside `spawn` blocks, checks that fluid bindings from the outer scope are not used across the thread boundary.

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

A phase error is *fatal*, and it is fatal early: the checker collects every problem it finds in one pass, prints the lot, and `clat` exits with status 1 without running a single statement. That is the trade you make for strict mode—you learn about all your phase mistakes at once, before anything happens:

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

4.5 First Encounter with the Phase Philosophy

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

4.5.1 The Lifecycle of Data

Most data in a program goes through a predictable lifecycle:

1. **Construction**: The data is built up incrementally. You add items to a list, set fields in a configuration, accumulate results.

2. **Stabilization**: At some point, the data is "done." The configuration is complete, the list has all its items, the computation is finished.

3. **Consumption**: The stabilized data is read, shared, serialized, sent to another thread, or stored for later use.

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

Lattice makes the boundary explicit:

Listing 4.19: The lifecycle of data in Lattice

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

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

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

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

4.5.2 The Chemistry Metaphor

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

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

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

- **Sublimation**: shallow freezing that locks the structure but leaves inner values mutable.
- **Bonds**: links between values that cause one to freeze when another does.
- **Reactions**: callbacks that fire when a value changes phase.
- **Pressure**: restrictions on what can change without fully freezing.

- **Alloys**: structs with per-field phase declarations.
- **Arenas**: memory regions where crystal values are stored.

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

4.5.3 Phase and Concurrency

The phase system's biggest payoff comes with concurrency. Lattice's `scope`/`spawn` structured concurrency model requires that values sent across a channel must be crystal, unphased, or scalar—fluid containers, sublimated values, and values containing closures, Refs, or iterators are rejected at runtime (see Section 20.1.2 for the full rules).

Listing 4.20: Phase and concurrency

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

flux data = [1, 2, 3]
// ch.send(data)
// error: can only send crystal (frozen) values on a channel

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

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

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

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

4.5.4 A Preview: The Forge Block

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

Listing 4.21: Forge: build fluid, end crystal

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

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

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

4.6 Practical Patterns

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

4.6.1 Accumulate Then Freeze

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

Listing 4.22: Accumulate then freeze

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

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

4.6.2 Configuration Objects

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

Listing 4.23: Configuration with defaults

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

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

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

    freeze(cfg)
    return cfg
}

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

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

4.6.3 The Fluid Working Copy

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

Listing 4.24: Working with a thawed copy

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

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

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

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

4.7 Exercises

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

2. **Strict Mode Workout.** Create a file with `#mode strict` at the top. Try using `let` to declare a variable. Try assigning to a `fix` binding. Record the error messages you see—and confirm that nothing in the file runs at all. Then freeze a value and read the original name afterwards, running that one under all three backends to watch the divergence this chapter warned you about.

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

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

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

What's Next

You now understand how Lattice thinks about variables and mutability. The `let`/`flux`/`fix` keywords give you explicit control over the lifecycle of your data, and the phase system enforces that control at runtime. We've barely scratched the surface of what the phase system can do—bonds, reactions, sublimation, pressure, and alloys await in Part III.

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

Part II

The Working Programmer

Chapter 5

Control Flow

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

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

Let's see what that looks like in practice.

5.1 if/else as Expressions

Here is the most basic conditional in Lattice:

Listing 5.1: A basic if/else

```
let temperature = 38

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

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

Listing 5.2: if/else as an expression

```
let temperature = 38

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

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

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

Expressions vs. Statements

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

5.1.1 How the Value Is Determined

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

Listing 5.3: Multi-line branches still produce values

```
let age = 25

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

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

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

5.1.2 What Happens Without an else?

If you omit the `else` branch, there is nothing for the false path to produce, and the expression evaluates to a "nothing" value:

Listing 5.4: A missing else leaves you with nothing

```
let temperature = 36

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

print(warning) // Output: nil   (unit under --tree-walk and --regvm)
```

An else-less if Is Not Portably nil

Which "nothing" you get depends on the backend. The default bytecode VM produces `nil`, so `warning == nil` is `true` and `warning ?? "none"` supplies the fallback. Under `--tree-walk` and `--regvm` the same expression produces `unit`—which is *not* equal to `nil` and which `??` does not treat as missing.

The portable habit is to write the `else` branch whenever you intend to use the result. An `if` with no `else` is a statement you happen to be allowed to bind, not a value-producing expression you should lean on.

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

5.1.3 Nested and Chained Conditions

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

Listing 5.5: Chained else if

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

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

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

When to Use if/else vs. match

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

5.1.4 Blocks as Expressions

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

Listing 5.6: Block expressions

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

print(hypotenuse) // Output: 5
```

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

5.2 Loops: while, loop, for

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

5.2.1 while Loops

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

Listing 5.7: Counting with while

```
flux count = 0

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

Notice the `flux` binding—we reassign `count` on every iteration, and `flux` declares that intent up front. (Phase in Lattice governs the *value*, not the binding, so rebinding a variable is a separate question from mutating what it holds—Chapter 4 draws that line precisely.) You can also write the update as `count += 1`; the compound assignment operators `+=`, `-=`, `*=`, `/=`, and `%=` work anywhere a plain assignment does.

Since v0.4.1 they also work on individual *elements*, including elements reached through a computed index:

Listing 5.8: Compound assignment on elements

```
flux totals = Map::new()
totals["widgets"] = 1
totals["widgets"] += 41
print(totals["widgets"]) // Output: 42

flux readings = [10, 20, 30]
let i = 0
readings[i + 1] += 5
print(readings) // Output: [10, 25, 30]
```

No ++ or --

Coming from C, Java, or JavaScript, your fingers may type `count++` on autopilot. Lattice has no increment or decrement operators—`count++` is rejected with `parse error: unexpected token '+' in expression`. Reach for `count += 1` instead.

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

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

Listing 5.9: Building a string with while

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

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

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

5.2.2 Infinite Loops with loop

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

Listing 5.10: An infinite loop with break

```
flux attempts = 0

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

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

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

Listing 5.11: A retry pattern with loop

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

5.2.3 for Loops and Iteration

The `for` loop iterates over collections and ranges:

Listing 5.12: Iterating over an array

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

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

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

Listing 5.13: Iterating over a range

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

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

Listing 5.14: Iterating with an index

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

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

Listing 5.15: Iterating over a string's characters

```
let greeting = "Hello"

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

The Loop Variable Is Fresh Each Iteration

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

5.2.4 Nested Loops

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

Listing 5.16: Nested loops for a multiplication table

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

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

5.3 break, continue, return

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

5.3.1 break — Exit the Loop

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

Listing 5.17: Finding the first negative number

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

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

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

break Outside a Loop

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

5.3.2 continue — Skip to the Next Iteration

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

Listing 5.18: Skipping negative readings

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

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

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

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

Listing 5.19: continue and break working together

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

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

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

5.3.3 return — Exit the Function

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

Listing 5.20: Early return from a function

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

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

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

Listing 5.21: Implicit return: no return keyword needed

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

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

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

Listing 5.22: Guard clauses with early return

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

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

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

break, continue, and Closures

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

5.4 Ranges

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

5.4.1 Creating Ranges

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

Listing 5.23: Half-open ranges

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

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

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

Range

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

5.4.2 Ranges with Expressions

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

Listing 5.24: Dynamic range endpoints

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

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

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

5.4.3 Ranges for Indexing and Slicing

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

Listing 5.25: Slicing with ranges

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

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

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

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

5.4.4 Ranges as First-Class Values

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

Listing 5.26: Ranges as first-class values

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

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

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

Listing 5.27: Using ranges for batch processing

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

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

5.5 The Nil-Coalescing Operator ??

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

Listing 5.28: Basic nil-coalescing

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

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

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

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

Nil-Coalescing Operator

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

Missing Map Keys and the Tree-Walker

On the default bytecode VM and on `--regvm`, indexing a `Map` with an absent key produces `nil`, which is exactly what `??` is looking for. The `--tree-walk` backend still produces `unit` for that same miss (v0.6.4), and `??` does not treat `unit` as missing—so `config["language"] ?? "en"` yields `()` there instead of `"en"`.
If a program has to behave identically on all three backends, reach for the `.get()` method, which returns `nil` for a missing key everywhere:

```
print(config.get("language") ?? "en") // Output: en  (all three backends)
```

Section 33.4.3 collects the remaining backend divergences in one place.

5.5.1 Short-Circuit Evaluation

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

Listing 5.29: Short-circuit evaluation in action

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

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

The compiler implements this with a conditional jump. It compiles the left operand, then emits `OP_JUMP_IF_NOT_NIL`—if the value on the stack is not nil, execution skips over the right operand entirely. Only if the value is nil does it pop the nil, evaluate the fallback, and continue. This means the

cost of a ?? expression when the left side is non-nil is a single comparison and branch—no function call overhead, no allocation.

5.5.2 Chaining ??

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

Listing 5.30: Chaining nil-coalescing operators

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

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

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

Listing 5.31: A practical configuration lookup

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

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

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

5.5.3 Combining ?? with Other Expressions

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

Listing 5.32: Nil-coalescing in various contexts

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

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

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

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

?? vs. if/else for Defaults

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

Constant Folding and ??

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

5.6 The Pipe Operator

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

Listing 5.33: Deeply nested function calls

```
fn strip_ends(s: String) -> String { s.trim() }
fn emphasize(s: String) -> String { "${s}!" }
fn shout(s: String) -> String { s.to_upper() }

let raw_input = "  hello world  "
let result = shout(emphasize(strip_ends(raw_input)))
print(result) // Output: HELLO WORLD!
```

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

Listing 5.34: Using pipe for clarity

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

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

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

pipe()

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

5.6.1 How pipe() Works

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

Listing 5.35: Step-by-step pipe transformation

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

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

This is equivalent to the following sequential code:

Listing 5.36: The same transformations without pipe

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

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

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

5.6.2 Building Data Pipelines

`pipe()` shines when processing collections:

Listing 5.37: A data processing pipeline

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

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

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

(A quick note on `reduce`: it takes the combining closure first and the starting accumulator second. Chapter 7 tours it properly.)

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

Listing 5.38: Adding a step to the pipeline

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

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

5.6.3 pipe() with Named Functions

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

Listing 5.39: Using named functions with pipe

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

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

Naming a Function Without Calling It

Writing `double` with no parentheses hands `pipe()` the function itself rather than its result. That works on the default bytecode VM and on `--regvm`; the `--tree-walk` backend rejects a bare function name with `undefined variable 'double'` (Section 6.1.1 shows the closure wrapper that works everywhere).

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

Listing 5.40: Combining named functions and closures in pipe

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

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

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

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

pipe() vs. Method Chaining

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

5.7 Putting It All Together

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

Listing 5.41: Comprehensive control flow example

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

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

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

    total = total + score
    count = count + 1

    let new_high = if highest == nil { score }
                   else if score > highest { score }
                   else { highest }
    highest = new_high

    let new_low = if lowest == nil { score }
                  else if score < lowest { score }
                  else { lowest }
    lowest = new_low
  }

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

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

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

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

This example exercises `for` loops, `continue` to skip invalid data, `if`/`else` as expressions for computing grades and min/max values, and `??` for safe defaults. Notice how every `if`/`else` chain produces a value directly—the running maximum is computed by an expression and then assigned, rather than built up with a nest of statements.

5.8 Exercises

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

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

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

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

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

What's Next?

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

Chapter 6

Functions and Closures

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

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

6.1 Defining Functions with fn

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

Listing 6.1: A basic function

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

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

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

Function Anatomy

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

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

6.1.1 Functions Are Values

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

Listing 6.2: Functions as values

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

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

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

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

A Tree-Walker Limitation

As of v0.6.4, referencing a named `fn` as a bare value (`let operation = add`) works on the default bytecode VM and on `--regvm`, but the `--tree-walk` backend rejects it with `undefined variable 'add'`—there, named functions can only be *called*. If you need first-class function values under the tree-walker, wrap the function in a closure: `let operation = |a, b| add(a, b)`. Section 33.4.3 catalogs the remaining backend differences.

6.1.2 Function Bodies as Expressions

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

Listing 6.3: Various function body styles

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

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

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

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

Lines That Continue the Previous Expression

Lattice statements are newline-separated, but a line that *starts* with `(`, `[`, or `|` is parsed as a continuation of the expression above it: a call, an index, or a pipe token. Had we written the final expression above as `(a_sq + b_sq)`, the parser would read the last two lines as `let b_sq = b` `* b(a_sq + b_sq)`—a call to `b`—and the program would die with `undefined variable 'b_sq'`. The same trap awaits a closure literal or an array literal used as a function's final expression. When the last expression of a body starts with one of these tokens, either drop the redundant delimiter or use an explicit `return`.

6.1.3 Recursive Functions

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

Listing 6.4: Recursive factorial

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

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

The VM has a call frame limit—`STACKVM_FRAMES_MAX`, 512 frames in v0.6.4 (`include/stackvm.h`)—to prevent runaway recursion from consuming all memory. Exceeding it raises `stack overflow (too many nested calls)`, which since v0.3.29 is an ordinary catchable runtime error rather than an abrupt exit:

Listing 6.5: Recursion depth is a catchable error

```
fn boom(n: Int) -> Int { boom(n + 1) }

let outcome = try { boom(0) } catch e { "caught: ${e.message}" }
print(outcome) // Output: caught: stack overflow (too many nested calls)
```

Each backend words the limit in its own way (`--tree-walk` reports `maximum recursion depth exceeded (limit: 1000)`, `--regvm` says `call stack overflow`), but all three raise something you can catch. Treat any of them as a signal to consider an iterative approach.

6.2 Type Annotations on Parameters and Return Types

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

Listing 6.6: Type annotations

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

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

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

6.2.1 How Runtime Type Checking Works

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

Listing 6.7: Type mismatch error

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

The any Type

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

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

6.2.2 Return Type Checking

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

Listing 6.8: Return type checking

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

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

6.2.3 Common Type Names

6.2.4 Spelling Out a Function's Shape

Bare `Fn` tells a reader that a parameter is callable, and nothing more. Since v0.3.29 you can write the whole signature:

Type Annotation	Matches
`Int`	Integer values
`Float`	Floating-point values
`String`	String values
`Bool`	`true` or `false`
`Array`	Array values
`Map`	Map values
`Set`	Set values
`Range`	Range values
`Fn`	Closure/function values
`Fn(A, B) -> C`	Closure values, with a documented shape
`any` / `Any`	Any type (no check)

Table 6.1: Built-in type annotations

Listing 6.9: A function-typed parameter

```
fn apply(f: Fn(Int) -> Int, x: Int) -> Int {
  f(x)
}

print(apply(|n| { n + 1 }, 41)) // Output: 42
```

The types inside `Fn(...)` are for the reader. Nothing verifies that the closure you hand over actually takes an `Int` and returns one—if it doesn't, you find out when the closure runs, or when the caller's own return-type check fires.

Bare `Fn` Checks More Than `Fn(...)`

Annotating a parameter `Fn` makes the VM verify at the call boundary that the argument really is callable: `apply(7, 41)` stops immediately with `function 'apply' parameter 'f' expects type Fn, got Int` on all three backends. The parenthesized form skips that boundary check on the default VM—the same call slides through and fails a step later. If you want the earliest possible error, bare `Fn` is the stricter annotation; if you want the signature documented, the parenthesized form says more to the next reader. Both parse and run everywhere.

Type Annotations Are Not Static Types

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

6.3 Default Parameter Values and Variadic Parameters

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

6.3.1 Default Parameter Values

A parameter can specify a default value using =:

Listing 6.10: Default parameter values

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

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

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

Default Values Are Computed Once

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

Parameters with defaults must come after required parameters:

Listing 6.11: Required parameters before defaults

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

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

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

6.3.2 Variadic Parameters

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

Listing 6.12: Variadic parameters

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

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

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

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

Listing 6.13: Variadic math functions

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

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

print(sum(1, 2, 3, 4, 5))         // Output: 15
print(average(85.0, 92.0, 78.0))  // Output: 85
```

reduce Takes the Closure First

The argument order is `reduce(combiner, initial_value)`—closure first, seed second. Swapping them is not a mistake the runtime catches cleanly: on the default VM a fold written seed-first quietly evaluates to `nil`, and only the surrounding return-type check (if there is one) reports anything at all. If a fold is mysteriously producing nothing, check the argument order first.

6.3.3 Combining Defaults and Variadics

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

Listing 6.14: Defaults and variadics together

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

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

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

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

6.4 Closures: `|x| x * 2`

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

Listing 6.15: Basic closure syntax

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

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

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

Listing 6.16: Multi-line closures

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

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

Closure Syntax

```
// Single expression
|params| expression

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

// No parameters -- note the space between the pipes
| | { body }
```

Closure parameters carry no type annotations. This keeps them lightweight for use in callbacks, iterators, and higher-order functions.

Two Rules That Point in Opposite Directions

Named functions *require* a type annotation on every parameter. Closures *forbid* them. Writing `|x: any| { x }` stops the parser cold with `parse error: expected ',', got ':'`—it has already committed to reading `x` as a parameter name and expects a comma or a closing pipe next.

The empty parameter list has its own trap. `||` is the logical-or operator, and the lexer claims it before the parser ever sees two pipes; `let f = || { 42 }` fails with `parse error: unexpected token '||' in expression`. Put a space between them—`| |`—and the closure parses.

The two rules sit next to each other more often than you would like, because a function that *takes* a closure annotates that parameter (`fn apply(f: Fn, x: Int)`) while the closure passed in annotates nothing.

6.4.1 Closures with Default and Variadic Parameters

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

Listing 6.17: Closures with defaults and variadics

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

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

6.4.2 Where Closures Shine

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

Listing 6.18: Closures in collection methods

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

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

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

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

let by_length = ["kilogram", "gram", "milligram"].sort_by(|a, b| a.len() - b.len())
print(by_length) // Output: [gram, kilogram, milligram]
```

A `sort_by` comparator returns a negative number, zero, or a positive number—the same convention as C's `qsort` or JavaScript's `Array.sort`. Note that it compares by *length* here rather than alphabetically: the relational operators < and > work on numbers only, so `a < b` on two strings raises `operands must be numbers for '<'`. For plain alphabetical order, `sort()` already knows how to order strings:

Listing 6.19: Sorting strings without a comparator

```
let sorted = ["banana", "apple", "cherry"].sort()
print(sorted) // Output: [apple, banana, cherry]
```

6.5 Closures Capture Their Environment

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

Listing 6.20: Closures capture enclosing variables

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

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

We bind the closure to `bump` and return that instead of writing the closure literal as the final expression—a line beginning with `|` would be swallowed as a continuation of the line above it, exactly the trap described earlier in this chapter.

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

Upvalue

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

6.5.1 How Upvalue Capture Works Under the Hood

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

1. **Compile time:** When the compiler sees a variable reference that doesn't resolve in the current function, it walks up the enclosing compiler chain via `resolve_upvalue`. If the variable is a local in the parent function, it marks that local as "captured" and records it with `is_local =`

`true`. If it's an upvalue of the parent, it chains through with `is_local = false`. The upvalue descriptors are emitted after the `OP_CLOSURE` instruction.

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

3. **Closing (when the scope exits):** When a function returns or a scope ends, the VM calls `close_upvalues`, which walks the open upvalue list. For each upvalue whose location is about to be popped from the stack, it copies the value into the `ObjUpvalue`'s `closed` field (via `value_clone_fast`) and redirects `location` to point at that field. The closure now reads from heap storage instead of the stack—and because closures that shared an `ObjUpvalue` still share it after closing, they keep seeing each other's writes.

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

Listing 6.21: Multiple closures sharing an upvalue

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

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

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

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

6.5.2 Mutating a Captured Container

Capturing an integer is one thing. Capturing an array and then *growing* it is where closures earn their keep—and where, until vo.6.4, Lattice quietly let you down.

Listing 6.22: A closure that accumulates into a captured array

```
flux collected = []
let record = |x| { collected.push(x) }

record("first")
record("second")

print(collected)       // Output: [first, second]
print(collected.len()) // Output: 2
```

This Used to Silently Do Nothing

Before vo.6.4, a mutating builtin method called on a captured receiver operated on a *copy* that was thrown away the instant the call returned. `collected.push(x)` appeared to work—no error, no warning—and `collected` stayed empty. Only scalar assignment (`count = count + 1`) and element assignment (`arr[i] = v`) wrote through to the captured cell.
That is why older Lattice code is full of pre-allocated arrays and hand-rolled index counters where a `push` would have read better. Those workarounds still run correctly; they are no longer necessary.

The fix is a dedicated opcode. When the compiler sees a method call whose receiver resolves to an upvalue rather than a local, it emits `OP_INVOKE_UPVALUE` instead of the usual "fetch a copy, then invoke" pair (`src/stackcompiler.c`; the register VM has an equivalent). At runtime the VM reaches through the `ObjUpvalue`'s `location` pointer and hands the builtin the real cell, so the mutation lands where the closure—and everyone else sharing that cell—can see it. The register backend and the tree-walker agree.

This makes the "bundle of closures over shared state" pattern behave the way you would expect. Here a factory hands back a map of closures that all read and write the same captured containers:

Listing 6.23: Closures in a map sharing mutable captured state

```
fn make_ledger() -> Map {
  flux entries = []
  flux totals = Map::new()

  let api = Map::new()
  api["record"] = |name, amount| {
    entries.push(name)
    totals.set(name, amount)
  }
  api["names"] = | | entries
  api["total_for"] = |name| totals[name]
  api
}

let ledger = make_ledger()
let record = ledger["record"]
let names = ledger["names"]
let total_for = ledger["total_for"]

record("coffee", 4)
record("books", 30)

print(names())            // Output: [coffee, books]
print(total_for("books")) // Output: 30
```

Array `push`/`pop`/`insert`/`remove_at`, map `set`/`remove`/`merge`, set `add`/`remove`/`clear`, and the buffer writers all participate. Element assignment through a captured container—`slots[i] = v`, `cfg[k] = v`—works as it always has.

Capture Does Not Defeat Freezing

A captured container that has been crystallized is still protected. Calling a mutating method on it raises the usual error, with the variable's name in the message:

```
flux collected = [1, 2]
freeze(collected)
let record = |x| { collected.push(x) }
record(3)
// vm error: cannot call mutating method 'push' on crystal value
//           'collected' (use thaw(collected) to make it mutable)
```

Phases are a property of the value, and a closure holding a reference to that value inherits the same rules. Chapter II works through the phase system in detail.

6.5.3 Closures in Loops

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

Listing 6.24: Closures in loops capture correctly

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

for m in multipliers {
  print(m(10))
}
// Output: 10 20 30 40   (1*10, 2*10, 3*10, 4*10)
```

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

You Cannot Call an Indexed Closure Directly

It is tempting to write `multipliers[0](10)`, but the parser rejects it with `parse error: expected ',', got '('`. A call may only follow a plain identifier, so an index expression—or a map lookup, or any other subscript—has to be bound to a name first:

```
let first = multipliers[0]
print(first(10)) // Output: 10
```

The same restriction is why closures fetched out of a `Map` get a `let` binding before they are called.

Closures over flux Variables in while Loops

In a `while` loop, a `flux` counter variable is *not* rebound each iteration—it's the same mutable slot throughout. Closures created in such a loop all share it, and they all see its final value:

```
flux callbacks = []
flux i = 0
while i < 5 {
  callbacks.push(|x| x * i)
  i = i + 1
}
for c in callbacks { print(c(10)) }
// Output: 50 50 50 50 50   -- every closure sees i == 5
```

If you need per-iteration capture in a `while` loop, create a `let` binding inside the loop body:

```
flux callbacks = []
flux i = 0
while i < 5 {
  let captured_i = i  // fresh binding each iteration
  callbacks.push(|x| x * captured_i)
  i = i + 1
}
for c in callbacks { print(c(10)) }
// Output: 0 10 20 30 40
```

6.6 Higher-Order Functions

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

6.6.1 Functions That Return Functions

Listing 6.25: A function factory

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

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

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

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

6.6.2 Callbacks and Event Handlers

Higher-order functions are natural for callback patterns:

Listing 6.26: Processing with callbacks

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

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

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

6.6.3 When a Callback Fails

Handing someone else's code a closure raises an obvious question: what happens when the closure blows up halfway through the job? As of v0.6.4 the answer is the one you want—the error travels out of the callback, out of the higher-order function, and into whatever `try` block is waiting for it.

Listing 6.27: An error inside a callback reaches the caller's catch

```
fn apply_all(items: Array, transform: Fn) -> Array {
  flux out = []
  for item in items {
    out.push(transform(item))
  }
  out
}

let safe = try {
  apply_all([4, 2, 1], |n| 12 / n)
} catch err {
  "pipeline failed: ${err.message}"
}
print(safe) // Output: [3, 6, 12]

let broken = try {
  apply_all([4, 0, 1], |n| 12 / n)
} catch err {
  "pipeline failed: ${err.message}"
}
print(broken) // Output: pipeline failed: division by zero
```

Nothing in `apply_all` mentions error handling. The division by zero unwinds through the closure call, through the `for` loop, through the function frame, and lands in the caller's `catch`—with the half-built `out` array released on the way. Built-in higher-order methods behave the same way:

Listing 6.28: Errors propagate out of a map callback

```
let result = try {
  [1, 0, 2].map(|n| 12 / n)
} catch err {
  "map failed: ${err.message}"
}
print(result) // Output: map failed: division by zero
```

Error Messages Carry No Backend Prefix

The text you see in `err.message` is the bare message—`"division by zero"`—on all three backends. The prefixes that appear on *uncaught* errors (`vm error:` on the default VM, `error:` under `--tree-walk`, `regvm error:` under `--regvm`) are added by the top-level reporter, not stored on the error value.

6.6.4 Function Composition

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

Listing 6.29: Composing functions

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

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

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

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

Listing 6.30: pipe for left-to-right composition

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

6.6.5 Building Abstractions with Higher-Order Functions

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

Listing 6.31: Abstracting the retry pattern

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

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

Listing 6.32: A memoization wrapper

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

let slow_square = |x| {
  print("Computing ${x}^2...")
  x * x
}

let fast_square = memoize(slow_square)
print(fast_square(5))
// Output: Computing 5^2...
//         25
print(fast_square(5))
// Output: 25          (cached -- no "Computing" line)
print(fast_square(3))
// Output: Computing 3^2...
//         9
```

The `memoize` function creates a closure that wraps the original function with a cache. The cache is a captured `flux` map shared across all calls to the memoized function—and, as Section 6.5.2 showed, writes into it stick.

6.7 require and ensure — Contracts on Your Functions

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

6.7.1 Preconditions with require

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

Listing 6.33: require preconditions

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

print(withdraw(100.0, 30.0)) // Output: 70
withdraw(100.0, -5.0)
// Error: require failed in 'withdraw': withdrawal amount must be positive
```

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

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

6.7.2 Postconditions with ensure

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

Listing 6.34: ensure postconditions

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

print(abs_val(-42.0)) // Output: 42
print(abs_val(7.0))   // Output: 7
```

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

Listing 6.35: Multiple contracts on a function

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

print(divide(10.0, 3.0))  // Output: 3.33333
divide(10.0, 0.0)
// Error: require failed in 'divide': cannot divide by zero
```

6.7.3 When to Use Contracts

Contracts work best as:

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

Contracts vs. Error Handling

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

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

6.8 Putting It All Together

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

Listing 6.36: A validated pipeline builder

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

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

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

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

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

let temperatures = [0.0, 25.0, 100.0, 150.0, -40.0]
for temp in temperatures {
  let converted = celsius_to_fahrenheit(temp)
  print("${temp}C = ${converted}F")
}
// Output:
// 0C = 32F
// 25C = 77F
// 100C = 212F
// 150C = 212F
// -40C = 32F
```

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

Why Not Call It `clamp`?

`clamp` is already a built-in numeric function. On the bytecode backends a user-defined `fn clamp` shadows it without complaint, but under `--tree-walk` the built-in wins and the program fails with an arity complaint from the builtin: `clamp() expects (Int|Float, Int|Float, Int|Float)`. Naming a function after a builtin is a portability hazard—pick a name the runtime hasn't claimed.

6.9 Exercises

1. **once().** Write a function `fn once(f: Fn) -> Fn` that returns a wrapper which calls `f` on the first invocation and returns the cached result on subsequent calls.

2. **Partial Application.** Write a function `fn partial(f: Fn, first_arg: any) -> Fn` that returns a new function with the first argument pre-filled. Given `let add = |a, b| a + b`, binding `let add_five = partial(add, 5)` should make `add_five(3)` return `8`. (Remember that a call can only follow a plain name, so the result has to be bound before it is called.)

3. **Validated List.** Write a function `fn make_list(validator: Fn) -> Map` that returns a map with `"add"` and `"items"` closures. The `"add"` closure should only add items that pass the validator. Use upvalue capture to share state between the closures.

4. **Function Pipeline with Error Handling.** Extend the pipeline pattern by writing `fn safe_pipeline(...transforms: Fn) -> Fn` that wraps each transform in a try/catch. If any transform fails, the pipeline should return the error message instead of crashing.

5. **Contract Challenge.** Write a function `fn binary_search` that searches a sorted array for a value. Add `require` contracts to verify the array is sorted and the target is of the correct type. Add an `ensure` contract to verify that if the function returns a non-negative index, the element at that index equals the target.

What's Next?

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

Chapter 7

Arrays, Maps, Sets, and Friends

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

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

7.1 Arrays: Creation, Indexing, and Mutation

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

Listing 7.1: Creating arrays

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

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

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

How Arrays Print

When `print` renders an array, string elements appear without quotation marks: `["a", "b"]` prints as `[a, b]`. We show the actual output in the comments throughout this chapter.

7.1.1 Indexing and Slicing

Elements are accessed by zero-based index:

Listing 7.2: Array indexing

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

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

Indexing outside the array's bounds is a runtime error on every backend, and the message names both the offending index and the current length:

Listing 7.3: Out-of-bounds indexing

```
let colors = ["red", "green", "blue", "yellow"]
print(colors[7])
// vm error: array index out of bounds: 7 (len 4)
```

The `vm error:` prefix is the default stack VM speaking; `--tree-walk` and `--regvm` word the same refusal slightly differently. We'll come back to that in Section 33.4.3.

No Negative Array Indices

Unlike Python, Lattice arrays do not support negative indices: `colors[-1]` raises `array index out of bounds: -1 (len 4)` on the default VM, and the tree-walker reports the same refusal with the index printed as a very large unsigned number. (As of v0.6.4 the register backend still accepts `-1` as "count from the end"—treat that as an accident of implementation, not a feature.) To reach the end of an array, use `colors.last()` or `colors[colors.len() - 1]`.

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

Listing 7.4: Array slicing

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

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

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

7.1.2 Mutation with push and pop

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

Listing 7.5: push and pop

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

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

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

The `push` method appends an element to the end, and `pop` removes and returns the last element (or `nil` if the array is empty). Both methods respect phase constraints—as of v0.5.0, every backend enforces the crystal phase, so calling a mutating method on a frozen array is a runtime error with a helpful hint:

Listing 7.6: Mutating a crystal array is an error

```
fix pinned = ["apple", "banana"]
pinned.push("cherry")
// vm error: cannot call mutating method 'push' on crystal
//   value 'pinned' (use thaw(pinned) to make it mutable)
```

We'll dig into phases properly in Chapter 11; for now, remember that a crystal array is solid all the way through—no atoms move until you `thaw` it.

Listing 7.7: Building arrays with push

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

7.1.3 Mutating Nested Elements

Collections nest freely—arrays of arrays, maps of arrays, and so on. As of v0.6.4, calling a mutating method on a nested element does what it says on every backend:

Listing 7.8: Mutating an element inside a container

```
flux grid = [[1], [2, 3]]
grid[0].push(9)
print(grid)          // Output: [[1, 9], [2, 3]]
print(grid[0].len()) // Output: 2

flux stats = Map::new()
stats["hits"] = [1]
stats["hits"].push(2)
print(stats["hits"]) // Output: [1, 2]
```

This deserves a historical footnote: before v0.6.4, the bytecode backends would clone the element, mutate the clone, and silently drop it—`grid[0].push(9)` was a no-op. Older Lattice code often worked around this by reading the element into a variable, mutating it, and writing it back. That workaround still runs, but you no longer need it. Under the hood the compilers now emit a dedicated `OP_INVOKE_MUT` (and `ROP_INVOKE_MUT` on the register backend) that writes the mutated element back into its parent.

Freezing the outer container seals the inner ones too. The refusal names the method but not a variable, because the receiver is an element rather than a binding:

Listing 7.9: A frozen parent seals its elements

```
flux grid = [[1], [2, 3]]
freeze(grid)
grid[0].push(9)
// vm error: cannot call mutating method 'push' on a frozen value
```

Compound assignment reaches through indices too—including computed ones. This has worked since v0.4.1; before that, the same line crashed the interpreter:

Listing 7.10: Compound assignment on elements

```
flux totals = [10, 20, 30]
let i = 0
totals[i + 1] += 5
print(totals) // Output: [10, 25, 30]
```

Two details make this safe in v0.6.4 that were not safe before. The index expression `i + 1` is evaluated exactly once, so a side-effecting index can no longer land on the wrong element; and a frozen target still refuses, so `totals[0] += 1` after `freeze(totals)` raises `cannot modify a frozen value` rather than quietly succeeding.

7.1.4 Essential Non-Closure Methods

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

Listing 7.11: Array inspection methods

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

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

Listing 7.12: Array transformation methods

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

print(numbers.reverse())  // Output: [6, 2, 9, 5, 1, 4, 1, 3]
print(numbers.unique())   // Output: [3, 1, 4, 5, 9, 2, 6]
print(numbers.sort())     // Output: [1, 1, 2, 3, 4, 5, 6, 9]
print(numbers.sum())      // Output: 31
print(numbers.min())      // Output: 1
print(numbers.max())      // Output: 9
```

Listing 7.13: Array subsetting methods

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

print(alphabet.take(3))   // Output: [a, b, c]
print(alphabet.drop(4))   // Output: [e, f, g]
print(alphabet.chunk(3))  // Output: [[a, b, c], [d, e, f], [g]]
```

Listing 7.14: join, enumerate, and zip

```
let words = ["Lattice", "is", "great"]
print(words.join(" "))  // Output: Lattice is great
print(words.join("-"))  // Output: Lattice-is-great

let items = ["apple", "banana", "cherry"]
print(items.enumerate())
// Output: [[0, apple], [1, banana], [2, cherry]]

let keys = ["name", "age", "city"]
let vals = ["Alice", 30, "Paris"]
print(keys.zip(vals))
// Output: [[name, Alice], [age, 30], [city, Paris]]
```

sort() vs. sort_by()

The `sort()` method (no closure) works only with homogeneous arrays of `Int`, `Float`, or `String`. It uses the natural ordering (ascending numeric, lexicographic for strings). For custom sorting logic or heterogeneous data, use `sort_by()` with a comparator closure.

7.2 Closure-Based Array Methods

The real power of Lattice's arrays comes from methods that accept closures. These let you express transformations declaratively—saying *what* you want rather than *how* to compute it.

7.2.1 map — Transform Every Element

`map` applies a closure to each element and collects the results:

Listing 7.15: Mapping over arrays

```
let prices = [10.0, 25.0, 7.5, 42.0]

let with_tax = prices.map(|p| p * 1.08)
print(with_tax) // Output: [10.8, 27, 8.1, 45.36]

let labels = prices.map(|p| "$${p}")
print(labels) // Output: [$10, $25, $7.5, $42]
```

The original array is untouched—`map` always returns a new array. Look closely at the outputs: whole-number floats like `27.0` render without the trailing `.0`, both in `print` and in string interpolation. The values are still floats—only their display is compact.

7.2.2 filter — Keep Matching Elements

`filter` returns a new array containing only elements for which the closure returns `true`:

Listing 7.16: Filtering arrays

```
let temperatures = [36.5, 37.2, 38.8, 36.9, 39.1, 37.0]

let fevers = temperatures.filter(|t| t > 37.5)
print(fevers) // Output: [38.8, 39.1]

let words = ["hello", "a", "Lattice", "is", "wonderful"]
let long_words = words.filter(|w| w.len() > 3)
print(long_words) // Output: [hello, Lattice, wonderful]
```

7.2.3 reduce — Collapse to a Single Value

`reduce` folds an array into a single value using an accumulator:

Listing 7.17: Reducing arrays

```
let numbers = [1, 2, 3, 4, 5]

// Sum: the closure comes first, the initial value second
let sum = numbers.reduce(|acc, n| acc + n, 0)
print(sum) // Output: 15

// Product
let product = numbers.reduce(|acc, n| acc * n, 1)
print(product) // Output: 120

// Build a string
let words = ["the", "quick", "brown", "fox"]
let sentence = words.reduce(|acc, w| {
  if acc == "" { w } else { "${acc} ${w}" }
}, "")
print(sentence) // Output: the quick brown fox
```

The first argument to `reduce` is the closure; the second is the initial accumulator value. The closure receives two arguments: the current accumulator and the next element.

Both arguments are required on the default VM and under `--tree-walk`. Drop the initial value and the default VM answers `type 'Array' has no method 'reduce'` while the tree-walker answers `.reduce() expects 2 arguments (closure, initial_value)`. The register backend is the odd one out:

`[1, 2, 3].reduce(|acc, n| acc + n)` returns `6` there, seeding the accumulator from the first element. Treat that as a divergence rather than a feature, and pass both arguments so your fold runs everywhere.

7.2.4 sort_by — Custom Sorting

`sort_by` sorts an array using a comparator closure. The closure takes two elements and should return a negative number if the first should come before the second, positive if after, and zero if equal:

Listing 7.18: Custom sorting

```
let people = [
  ["Alice", 30],
  ["Charlie", 25],
  ["Bob", 35]
]

// Sort by age (second element)
let by_age = people.sort_by(|a, b| a[1] - b[1])
print(by_age)
// Output: [[Charlie, 25], [Alice, 30], [Bob, 35]]

// Sort names by length
let names = ["Alice", "Bob", "Charlie", "Jo", "Eve"]
let by_length = names.sort_by(|a, b| a.len() - b.len())
print(by_length) // Output: [Jo, Bob, Eve, Alice, Charlie]

// Descending order: swap the operands
let scores = [72, 95, 61, 88]
let descending = scores.sort_by(|a, b| b - a)
print(descending) // Output: [95, 88, 72, 61]
```

Comparison Operators Don't Order Strings

The operators < and > work only on numbers—`"apple" < "banana"` raises `operands must be numbers for '<'` on the default VM, and an equivalent refusal on the other two backends. For lexicographic string order, use plain `sort()`, which handles `String` arrays natively and compares the full byte sequence. Inside a `sort_by` comparator, stick to numeric sort keys.

7.2.5 group_by — Categorize Elements

`group_by` partitions an array into a map based on a key function:

Listing 7.19: Grouping elements

```
let words = ["apple", "avocado", "banana", "blueberry", "cherry", "coconut"]

let by_first_letter = words.group_by(|w| w[0..1])
print(by_first_letter)
// Output: {"c": [cherry, coconut], "b": [banana, blueberry],
//          "a": [apple, avocado]}

let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let parity = numbers.group_by(|n| if n % 2 == 0 { "even" } else { "odd" })
print(parity)
// Output: {"even": [2, 4, 6, 8, 10], "odd": [1, 3, 5, 7, 9]}
```

Notice the group order: `"c"` before `"a"`. The result is a `Map`, and maps order their entries by internal hash layout, not by insertion (Section 7.5.1).

7.2.6 More Closure Methods

Listing 7.20: find, any, all

```
let inventory = [
  ["widget", 150],
  ["gadget", 0],
  ["doohickey", 42],
  ["thingamajig", 7]
]

// find: first matching element
let out_of_stock = inventory.find(|item| item[1] == 0)
print(out_of_stock) // Output: [gadget, 0]

// find_index: index of first match
let idx = inventory.find_index(|item| item[0] == "doohickey")
print(idx) // Output: 2

// any: at least one match?
let has_low_stock = inventory.any(|item| item[1] < 10)
print(has_low_stock) // Output: true

// all: every element matches?
let all_in_stock = inventory.all(|item| item[1] > 0)
print(all_in_stock) // Output: false
```

Both `any` and `all` short-circuit: `any` stops at the first `true`, and `all` stops at the first `false`. `find_index` and `partition` (below) joined the vocabulary in v0.3.29, alongside the map-level `any`/`all` we'll meet in Section 7.5.1.

Listing 7.21: flat_map, each, and partition

```
// flat_map: map then flatten one level
let sentences = ["hello world", "good morning"]
let words = sentences.flat_map(|s| s.split(" "))
print(words) // Output: [hello, world, good, morning]

// each: side effects only (returns unit)
let items = [1, 2, 3]
items.each(|n| print("Item: ${n}"))
// Output: Item: 1 \n Item: 2 \n Item: 3

// partition: split into [matching, non-matching]
let nums = [1, 2, 3, 4, 5, 6, 7, 8]
let result = nums.partition(|n| n % 2 == 0)
print(result) // Output: [[2, 4, 6, 8], [1, 3, 5, 7]]
```

One parser quirk worth knowing: an array literal at the start of a statement can be misread as an index into the previous expression, so `[1, 2, 3].each(...)` directly after another statement is a parse error (`expected ']', got ','`). Binding the array to a variable first, as above, sidesteps the ambiguity.

`each` and `flatten` Are VM-Only

Two array methods in this chapter are missing from the tree-walking interpreter in v0.6.4: `.each()` and `.flatten()` both raise `unknown method` under `--tree-walk`, though they work on the default stack VM and on `--regvm`. A plain `for` loop replaces `each` everywhere, and `flat_map(|x| x)` replaces `flatten`.

7.2.7 When a Callback Fails

A closure handed to a combinator is ordinary Lattice code, and it can fail like ordinary Lattice code. For `map`, v0.6.4 gives the answer you want on every backend: the failure propagates out of the combinator as a catchable error—no silent `nil` results, no half-built arrays:

Listing 7.22: Errors escape from a map callback

```
let nums = [1, 0, 2]
try {
  let doubled = nums.map(|x| 12 / x)
  print(doubled)
} catch e {
  print("caught: ${e.message}")
}
// Output: caught: division by zero
```

Only `map` Unwinds Everywhere

`map` is the one array combinator whose callback failure is guaranteed to reach your handler on all three backends. The others do not unwind on the bytecode engines: `filter`, `reduce`, `any`, `all`, `find`, `find_index`, `flat_map`, `partition`, `group_by`, and `sort_by` all end the process on the default stack VM and on `--regvm`—no output, no diagnostic, and the surrounding `catch` never runs. Under `--tree-walk` every one of them recovers and prints `caught: division by zero`. (`each` behaves the same way on the two VMs; the tree-walker does not implement it at all, per the previous warning.)

Section 26.5 records the same split for lazy `iter(...)` chains. Until the backends are harmonized, validate the data *before* the pipeline rather than catching after it.

Chaining Methods

You can chain closure methods to build expressive data transformations:

```
let result = data
  .filter(|x| x > 0)
  .map(|x| x * 2)
  .sort_by(|a, b| a - b)
  .take(5)
```

Each method returns a new array, so the chain flows naturally from left to right.

7.3 Non-Closure Array Methods Reference

Method	Returns	Description
`.len()`	`Int`	Number of elements
`.contains(v)`	`Bool`	Whether array contains `v`
`.index_of(v)`	`Int`	Index of first `v`, or `-1`
`.first()`	`Any`	First element (or `unit`)
`.last()`	`Any`	Last element (or `unit`)
`.reverse()`	`Array`	New reversed array
`.unique()`	`Array`	New array without duplicates
`.sort()`	`Array`	New sorted array (natural order)
`.flatten()`	`Array`	Flatten one level of nesting
`.join(sep)`	`String`	Elements joined by separator
`.zip(other)`	`Array`	Pairs with another array
`.enumerate()`	`Array`	`[index, value]` pairs
`.take(n)`	`Array`	First `n` elements
`.drop(n)`	`Array`	Skip first `n` elements
`.chunk(n)`	`Array`	Sub-arrays of size `n`
`.sum()`	`Number`	Sum of numeric elements
`.min()`	`Number`	Smallest element
`.max()`	`Number`	Largest element
`.push(v)`	`Unit`	Append element (mutates)
`.pop()`	`Any`	Remove and return last (mutates)

Table 7.1: Non-closure array methods

7.4 The Spread Operator ...

The spread operator `...` unpacks an array into another array:

Listing 7.23: Spread operator basics

```
let front = [1, 2, 3]
let back = [7, 8, 9]

let combined = [...front, 4, 5, 6, ...back]
print(combined) // Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Under the hood, when the compiler sees a spread expression (`EXPR_SPREAD`) inside an array literal, it emits the elements normally, builds the array with `OP_BUILD_ARRAY`, and then emits `OP_ARRAY_FLATTEN` to flatten one level of nesting. This means spread elements are expanded into the containing array.

Listing 7.24: Common spread patterns

```
// Prepending and appending
let items = [2, 3, 4]
let with_bookends = [1, ...items, 5]
print(with_bookends) // Output: [1, 2, 3, 4, 5]

// Merging multiple arrays
let a = [1, 2]
let b = [3, 4]
let c = [5, 6]
let merged = [...a, ...b, ...c]
print(merged) // Output: [1, 2, 3, 4, 5, 6]

// Creating a copy with modifications
let original = ["red", "green", "blue"]
let extended = [...original, "yellow", "purple"]
print(extended) // Output: [red, green, blue, yellow, purple]
```

Spread Only Works in Array Literals

The spread operator `...` works inside array literal brackets `[...]`. It is not a general-purpose operator that can be used in arbitrary expressions. For function arguments, the variadic `...` syntax has a different meaning (see Section 6.3.2).

7.5 Maps

A map (also known as a dictionary or hash table) stores key-value pairs. In Lattice, maps are created with `Map::new()` and populated via index assignment:

Listing 7.25: Creating and populating maps

```
let user = Map::new()
user["name"] = "Alice"
user["age"] = 30
user["email"] = "alice@example.com"

print(user["name"])  // Output: Alice
print(user["age"])   // Output: 30
```

Map Keys

Map keys in Lattice are strings internally. When you use a non-string value as a key, it's converted to its string representation. This means `m[42]` and `m["42"]` refer to the same entry. One byte-level caveat: keys are stored as C strings, so a key containing an embedded NUL byte is truncated at the NUL (Section 8.5).

7.5.1 Map Methods

Listing 7.26: Core map methods

```
let config = Map::new()
config["host"] = "localhost"
config["port"] = 8080
config["debug"] = true

// Checking keys
print(config.has("host"))     // Output: true
print(config.has("timeout"))  // Output: false

// Safe access with .get()
print(config.get("host"))     // Output: localhost
print(config.get("timeout"))  // Output: nil

// Size
print(config.len()) // Output: 3

// .set() is the method form of index assignment
config.set("retries", 3)
print(config.get("retries")) // Output: 3
```

Index assignment and `.set()` are interchangeable ways to add or overwrite an entry. `.set()` returns `unit`, so it doesn't chain—call it for its effect. (Since v0.6.4 the mutating map methods also work on a temporary, so `Map::new().set("a", 1)` is accepted everywhere instead of failing on the tree-walker. Because `.set()` hands back `unit`, the result of that expression is `unit`, not the map.)

Use `remove`, Not `delete`

The register backend carries an undocumented `.delete(key)` alias for `.remove(key)`. The other two backends do not: the default VM answers `type 'Map' has no method 'delete'` and the tree-walker `unknown method '.delete()' on Map`. Write `.remove()` and your program runs everywhere.

Maps also have their own `any` and `all` (added in v0.3.29), mirroring the array versions. The callback receives *two* arguments—the key and the value:

Listing 7.27: any and all over map entries

```
flux scores = Map::new()
scores["Alice"] = 95
scores["Bob"] = 82

print(scores.any(|k, v| v > 90))  // Output: true
print(scores.all(|k, v| v > 90))  // Output: false
print(scores.all(|k, v| v > 50))  // Output: true
```

Listing 7.28: keys, values, and entries

```
let scores = Map::new()
scores["Alice"] = 95
scores["Bob"] = 82
scores["Charlie"] = 91

print(scores.keys())    // Output: [Charlie, Bob, Alice]
print(scores.values())  // Output: [91, 82, 95]
print(scores.entries())
// Output: [[Charlie, 91], [Bob, 82], [Alice, 95]]
```

Maps Are Not Ordered

Alice went in first but Charlie came out first. The order of `keys()`, `values()`, and `entries()` follows the map's internal hash layout—deterministic for a given set of keys, but *not* insertion order. If you need a specific order, sort the keys yourself.

Listing 7.29: Removing keys and merging maps

```
let defaults = Map::new()
defaults["theme"] = "light"
defaults["language"] = "en"
defaults["font_size"] = 14

let user_prefs = Map::new()
user_prefs["theme"] = "dark"
user_prefs["font_size"] = 16

// merge: user preferences override defaults
defaults.merge(user_prefs)
print(defaults["theme"])     // Output: dark
print(defaults["language"])  // Output: en
print(defaults["font_size"]) // Output: 16

// remove
defaults.remove("language")
print(defaults.has("language")) // Output: false
```

7.5.2 Updating Values In Place

Counters are the classic map workload, and compound assignment works directly on map elements (since v0.4.1—earlier releases crashed here, so you may find contorted read-modify-write code in old Lattice programs):

Listing 7.30: Compound assignment on map values

```
flux counts = Map::new()
counts["clicks"] = 1
counts["clicks"] += 41
print(counts["clicks"]) // Output: 42
```

The same goes for mutating methods on values stored in a map, as we saw in Section 7.1.3: `stats["hits"].push(2)` mutates the array inside the map.

You can iterate over a map's entries, keys, or values:

Listing 7.31: Iterating over maps

```
let inventory = Map::new()
inventory["apples"] = 50
inventory["oranges"] = 30
inventory["bananas"] = 45

// Iterate over entries
for entry in inventory.entries() {
  let item = entry[0]
  let count = entry[1]
  print("${item}: ${count} in stock")
}

// Iterate over keys
for key in inventory.keys() {
  print("We carry: ${key}")
}
```

7.5.3 Maps as Lightweight Objects

Until you learn about structs (Chapter 15), maps serve as a convenient way to group related data:

Listing 7.32: Maps as ad-hoc objects

```
fn create_point(x: Float, y: Float) -> Map {
  let point = Map::new()
  point["x"] = x
  point["y"] = y
  point
}

fn distance(p1: Map, p2: Map) -> Float {
  let dx = p1["x"] - p2["x"]
  let dy = p1["y"] - p2["y"]
  sqrt(dx * dx + dy * dy)
}

let origin = create_point(0.0, 0.0)
let target = create_point(3.0, 4.0)
print(distance(origin, target)) // Output: 5
```

Method	Returns	Description
`.len()`	`Int`	Number of key-value pairs
`.has(key)`	`Bool`	Whether key exists
`.get(key)`	`Any`	Value for key, or `nil`
`.set(key, v)`	`Unit`	Add or overwrite entry (mutates)
`.keys()`	`Array`	All keys as array
`.values()`	`Array`	All values as array
`.entries()`	`Array`	`[key, value]` pairs
`.any(fn)`	`Bool`	Does any `(k, v)` entry match?
`.all(fn)`	`Bool`	Do all `(k, v)` entries match?
`.remove(key)`	`Unit`	Remove key-value pair
`.merge(other)`	`Unit`	Merge another map (mutates)

Table 7.2: Map methods

7.6 Sets

A set is an unordered collection of unique elements. Lattice sets start empty with `Set::new()` and grow with `.add()` (an argument passed to `Set::new()` is ignored—there is no array-to-set constructor):

Listing 7.33: Creating sets

```
let colors = Set::new()
colors.add("red")
colors.add("green")
colors.add("blue")
colors.add("red")  // duplicate, ignored

print(colors)      // Output: Set{red, green, blue}
print(colors.len()) // Output: 3

colors.clear()      // added in v0.3.29
print(colors.len()) // Output: 0
```

7.6.1 Set Operations

Sets support the classic mathematical operations:

Listing 7.34: Set operations

```
let frontend = Set::new()
frontend.add("Alice")
frontend.add("Bob")
frontend.add("Charlie")

let backend = Set::new()
backend.add("Bob")
backend.add("Diana")
backend.add("Eve")

// Union: all members of either team
let all_devs = frontend.union(backend)
print(all_devs) // Output: Set{Charlie, Eve, Alice, Diana, Bob}

// Intersection: members of both teams
let fullstack = frontend.intersection(backend)
print(fullstack) // Output: Set{Bob}

// Difference: frontend-only developers
let frontend_only = frontend.difference(backend)
print(frontend_only) // Output: Set{Charlie, Alice}

// Symmetric difference: in one team but not both
let specialists = frontend.symmetric_difference(backend)
print(specialists) // Output: Set{Charlie, Eve, Alice, Diana}
```

As with maps, don't read anything into the order of the printed elements—sets are unordered, and the display order follows the internal hash layout.

7.6.2 Subset and Superset Testing

Listing 7.35: Subset and superset testing

```
let required_skills = Set::new()
required_skills.add("Python")
required_skills.add("SQL")

let candidate_skills = Set::new()
candidate_skills.add("Python")
candidate_skills.add("SQL")
candidate_skills.add("JavaScript")
candidate_skills.add("Docker")

print(required_skills.is_subset(candidate_skills)) // Output: true
print(candidate_skills.is_superset(required_skills)) // Output: true
```

7.6.3 Converting Between Sets and Arrays

Listing 7.36: Set conversion

```
let duplicates = [1, 2, 2, 3, 3, 3, 4]

// Quick deduplication via Set
let unique_set = Set::new()
for item in duplicates {
  unique_set.add(item)
}
let unique_array = unique_set.to_array()
print(unique_array) // Output: [1, 2, 3, 4]
```

Internally, sets are implemented as maps where the keys are the string representations of elements and the values are the actual `LatValue` objects. This means set membership testing is O(1) on average, the same as map key lookups.

7.7 Tuples

A tuple is a fixed-length, heterogeneous collection. Unlike arrays, tuples cannot grow or shrink after creation:

Method	Returns	Description
`.len()`	`Int`	Number of elements
`.has(v)`	`Bool`	Whether v is in the set
`.add(v)`	`Unit`	Add element (mutates)
`.remove(v)`	`Unit`	Remove element (mutates)
`.clear()`	`Unit`	Remove all elements (mutates)
`.to_array()`	`Array`	Convert to array
`.union(other)`	`Set`	Elements in either set
`.intersection(other)`	`Set`	Elements in both sets
`.difference(other)`	`Set`	Elements in self but not other
`.symmetric_difference(other)`	`Set`	Elements in one but not both
`.is_subset(other)`	`Bool`	All elements in other?
`.is_superset(other)`	`Bool`	Contains all of other?

Table 7.3: Set methods

Listing 7.37: Creating tuples

```
let point = (3.0, 4.0)
let record = ("Alice", 30, true)
let singleton = (42,)

print(point)   // Output: (3, 4)
print(record)  // Output: (Alice, 30, true)
```

Note the trailing comma in `(42,)`—this distinguishes a one-element tuple from a parenthesized expression.

7.7.1 Accessing Tuple Elements

Tuple elements are accessed by index:

Listing 7.38: Accessing tuple elements

```
let person = ("Alice", 30, "alice@example.com")

print(person[0])   // Output: Alice
print(person[1])   // Output: 30
print(person[2])   // Output: alice@example.com
print(person.len()) // Output: 3
```

Tuple Indexing Needs the Default VM

As of v0.6.4, reading a tuple element by index is implemented only on the default stack VM. The same `person[0]` raises `cannot index Tuple with Int` under `--tree-walk` and `cannot index Tuple` under `--regvm`. `.len()` works everywhere. If a program has to run on all three backends, carry the values in a small array instead.

7.7.2 Tuples vs. Arrays

When should you use a tuple instead of an array?

- Use **tuples** for fixed collections where the position carries meaning: coordinates `(x, y)`, key-value pairs `(key, value)`, function results that return multiple values.
- Use **arrays** for variable-length collections of similar items: a list of names, a sequence of measurements, a batch of records.

Listing 7.39: Tuples for multiple return values

```
fn min_max(numbers: Array) -> Tuple {
  flux lo = numbers[0]
  flux hi = numbers[0]
  for n in numbers {
    if n < lo { lo = n }
    if n > hi { hi = n }
  }
  (lo, hi)
}

let result = min_max([38, 12, 45, 7, 91, 23])
print("Min: ${result[0]}, Max: ${result[1]}")
// Output (default VM): Min: 7, Max: 91
```

The tuple-unpacking on the last line is the part that needs the default backend, per the warning above; the function itself is backend-agnostic.

Tuples are immutable by nature—think of them as pre-crystallized, and that is literal: `phase_of((3.0, 4.0))` answers `crystal`. You cannot push, pop, or reassign individual elements. An attempted write like `point[0] = 9` meets the phase guard first and is refused with `cannot modify a frozen value`—the same message on all three backends, differing only in the prefix each one puts in front of it. If you need to "update" a tuple, create a new one.

Bind the tuple with `flux` and the phase guard steps aside, but the write still fails one layer down—and there the backends part company on wording. `flux point = (3.0, 4.0)` followed by `point[0] = 9` gives `invalid index assignment` on the default VM, `cannot index into Tuple` under `--tree-walk`, and `cannot set index on Tuple` under `--regvm`. Three refusals, one outcome: the element does not change.

Listing 7.40: Tuples are compiled with OP_BUILD_TUPLE

```
// Under the hood, (1, "hello", true) compiles to:
// push 1, push "hello", push true, OP_BUILD_TUPLE 3
let metadata = (1, "hello", true)
print(metadata) // Output: (1, hello, true)
```

7.8 Choosing the Right Collection

Need	Use	Why
Ordered list, may grow	`Array`	Push/pop, index access
Key-value storage	`Map`	O(1) key lookup
Unique elements only	`Set`	Auto-deduplication
Fixed group of values	`Tuple`	Position-based meaning

Table 7.4: Collection type selection guide

Let's see these working together in a realistic scenario:

Listing 7.41: Collections working together

```
// Analyzing survey responses
let responses = [
  ["Alice", "Lattice", 5],
  ["Bob", "Python", 4],
  ["Charlie", "Lattice", 5],
  ["Diana", "Rust", 4],
  ["Eve", "Lattice", 3],
  ["Frank", "Python", 5],
  ["Grace", "Rust", 5]
]

// Group by language
let by_language = responses.group_by(|r| r[1])

// Collect unique languages
let languages = Set::new()
for response in responses {
  languages.add(response[1])
}

// Compute average rating per language
let averages = Map::new()
for lang in languages.to_array() {
  let group = by_language[lang]
  let total = group.reduce(|acc, r| acc + r[2], 0)
  averages[lang] = total / group.len()
}

print("Languages surveyed: ${languages}")
print("Average ratings:")
for entry in averages.entries() {
  print("  ${entry[0]}: ${entry[1]}")
}
// Output:
// Languages surveyed: Set{Python, Rust, Lattice}
// Average ratings:
//   Lattice: 4
//   Python: 4
//   Rust: 4
```

(All three averages land on 4 because `total` and `group.len()` are both integers, and integer division truncates: Lattice's 13/3 rounds down to 4.)

7.9 Exercises

1. **Flatten and Deduplicate.** Given a nested array like `[[1, 2], [2, 3], [3, 4]]`, write a one-liner using `.flatten()` and `.unique()` to produce `[1, 2, 3, 4]`.
2. **Word Frequency Counter.** Write a function that takes a string, splits it into words, and returns a map of word frequencies. Use `group_by` or manual map construction.
3. **Inventory System.** Build an inventory system using a map of item names to quantities. Write functions for `add_stock`, `remove_stock`, and `low_stock_report` (items with fewer than 10 units). Use `.filter()` and `.entries()` in your report function.
4. **Set Operations Challenge.** Given three sets representing students enrolled in Math, Science, and English classes, compute: (a) students taking all three, (b) students taking exactly one class, and (c) students taking Math but not Science.
5. **Data Pipeline.** Given an array of `[name, score]` pairs, write a pipeline using `filter`, `map`, `sort_by`, and `take` to produce a "Top 3 Scorers" leaderboard with formatted strings like `"1. Alice (95)"`.

What's Next?

We've toured Lattice's collection types and their rich method vocabularies. In Chapter 8, we'll zoom in on strings specifically—interpolation, escaping, case transformations, Unicode support, and regular expressions. Strings are Lattice's most method-rich type, and there's a lot to explore.

Chapter 8

Strings in Depth

Strings are the workhorses of most programs. They carry user messages, encode configuration, represent file paths, and compose network protocols. Lattice treats strings as first-class values with a rich set of built-in methods—over two dozen, from splitting and trimming to case transformations, character inspection, and byte-level surgery.

A Lattice string is a *counted sequence of bytes*. That phrasing matters more than it looks: since v0.6.4 the length is authoritative, so a string may contain NUL bytes and every method still does the right thing. We'll spend a whole section on that later in this chapter.

In this chapter, we'll go beyond the basics of string creation (covered in Chapter 3) and explore the full breadth of what Lattice strings can do.

8.1 Interpolation, Escaping, and Raw Strings

8.1.1 String Interpolation

Lattice strings support interpolation with the `${expr}` syntax. Any expression can appear inside the braces:

Listing 8.1: String interpolation

```
let name = "Lattice"
let version = 1.0

print("Welcome to ${name}!")           // Output: Welcome to Lattice!
print("Version: ${version}")           // Output: Version: 1
print("1 + 2 = ${1 + 2}")              // Output: 1 + 2 = 3
print("Name length: ${name.len()}")    // Output: Name length: 7
```

That second line is worth a pause. `version` really is the float `1.0`, but Lattice renders whole-number floats without the trailing `.0`—in interpolation and in `print` alike. The value hasn't changed type; only its display is compact.

Interpolation works in double-quoted strings and triple-quoted strings. The expression is evaluated, converted to a string, and spliced into the surrounding text. You can nest function calls, method chains, and even conditionals inside the braces:

Listing 8.2: Complex interpolation expressions

```
let scores = [85, 92, 78, 91]
let avg = scores.reduce(|a, b| a + b, 0) / scores.len()

print("Average: ${avg} (${if avg >= 90 { "A" } else { "B" }})")
// Output: Average: 86 (B)
```

How Deep Can Interpolation Nest?

Interpolated expressions may themselves contain interpolated strings, but not without limit: the parser caps interpolation nesting at 200 levels and reports `string interpolation nested too deep` beyond that. You will never hit this by accident; machine-generated source occasionally does.

8.1.2 Escape Sequences

Double-quoted strings support the escape sequences catalogued in Table 3.1: `\n`, `\t`, `\r`, `\\`, `\"`, `\'`, `\$` (a literal dollar sign, which suppresses interpolation), and `\xHH` for an arbitrary byte written as two hex digits.

Listing 8.3: Using escape sequences

```
print("Line 1\nLine 2")
// Output:
// Line 1
// Line 2

print("Tab\there")
// Output: Tab    here

print("She said \"hello\"")
// Output: She said "hello"

print("Hex: \x41\x42\x43")
// Output: Hex: ABC

print("dollar: \$notinterp")
// Output: dollar: $notinterp
```

The One Byte You Cannot Write

There is no `\0` escape, and `\x00` is not a way in either. Both truncate the literal at that point: `"A\0B".len()` is `1` and `"\0".len()` is `0`. This is a property of the *lexer*, not of strings—a Lattice string is perfectly happy to hold a NUL byte once one arrives by some other route. Section 8.5 shows the routes.

8.1.3 Single-Quoted Strings

Single-quoted strings work identically to double-quoted strings in Lattice—they support the same escape sequences and interpolation. The choice between `'...'` and `"..."` is purely stylistic.

8.1.4 Triple-Quoted Strings

For multi-line strings, triple quotes preserve whitespace and auto-dedent:

Listing 8.4: Triple-quoted strings

```
let poem = """
    Roses are red,
    Violets are blue,
    Lattice is fast,
    And expressive too.
    """

print(poem)
// Output:
// Roses are red,
// Violets are blue,
// Lattice is fast,
// And expressive too.
```

Triple-quoted strings automatically strip the common leading whitespace from all lines, so you can indent them naturally in your code. They also support interpolation and escape sequences.

8.2 Core String Methods

8.2.1 split — Break a String Apart

The `split` method divides a string by a delimiter:

Listing 8.5: Splitting strings

```
let csv_line = "Alice,30,Engineer"
let fields = csv_line.split(",")
print(fields) // Output: [Alice, 30, Engineer]

let words = "the quick brown fox".split(" ")
print(words) // Output: [the, quick, brown, fox]

// Empty delimiter splits into individual bytes
let chars = "hello".split("")
print(chars) // Output: [h, e, l, l, o]
```

(Arrays print their string elements without quotation marks, which is why the output looks barer than the input.)

The Edges of split

Splitting is one of those operations whose interesting behavior lives entirely at the boundaries. Lattice's rules, pinned as of v0.6.4 and identical on all three backends:

Listing 8.6: split edge cases

```
print("".split(",").len())     // Output: 0   -- empty input, no parts at all
print("ab,".split(","))        // Output: [ab, ]
print("ab,".split(",").len())  // Output: 2   -- trailing separator leaves an empty
    part
print(",ab".split(","))        // Output: [, ab]
print(",ab".split(",").len())  // Output: 2   -- and so does a leading one

print("aaa".split("aa"))       // Output: [, a]
print("aaa".split("aa").len()) // Output: 2
```

That last pair is the one to memorize. `split` consumes a separator once it matches and resumes *after* it, so `"aaa"` yields the empty part before the match and the leftover `"a"` after it—two parts, not three.

A Genuine Behavior Change

Before v0.6.4, `"aaa".split("aa")` returned three parts, because the scan advanced one byte at a time and found a second, overlapping match. As of v0.6.4 separators are non-overlapping and the answer is two. If you have code that leaned on the old count, this is the release that changes its mind.

8.2.2 trim, trim_start, trim_end

These methods remove whitespace from string edges:

Listing 8.7: Trimming whitespace

```
let messy = "   hello, world!   "

// Brackets make the surviving spaces visible
print("[" + messy.trim() + "]")       // Output: [hello, world!]
print("[" + messy.trim_start() + "]") // Output: [hello, world!   ]
print("[" + messy.trim_end() + "]")   // Output: [   hello, world!]
```

Whitespace includes spaces, tabs, newlines, and carriage returns.

8.2.3 replace

`replace` substitutes all occurrences of a substring:

Listing 8.8: String replacement

```
let template = "Hello, {name}! Welcome to {place}."

let greeting = template
  .replace("{name}", "Alice")
  .replace("{place}", "Lattice")

print(greeting) // Output: Hello, Alice! Welcome to Lattice.

// Replace all occurrences
let stuttered = "b-b-banana"
print(stuttered.replace("b-", "")) // Output: banana
```

8.2.4 contains, starts_with, ends_with

These methods test whether a string contains, begins with, or ends with a given substring:

Listing 8.9: String searching

```
let filename = "report_2024.pdf"

print(filename.contains("2024"))     // Output: true
print(filename.starts_with("report")) // Output: true
print(filename.ends_with(".pdf"))    // Output: true
print(filename.ends_with(".csv"))    // Output: false
```

8.2.5 count

`count` returns the number of non-overlapping occurrences of a substring:

Listing 8.10: Counting occurrences

```
let text = "abracadabra"

print(text.count("a"))   // Output: 5
print(text.count("abra")) // Output: 2
print(text.count("z"))   // Output: 0
```

8.3 Case Transformations

Lattice provides a complete set of case transformation methods, useful for normalizing identifiers, formatting output, and converting between naming conventions.

Listing 8.11: Basic case conversion

```
let greeting = "Hello, World!"

print(greeting.to_upper()) // Output: HELLO, WORLD!
print(greeting.to_lower()) // Output: hello, world!
```

8.3.1 Naming Convention Converters

These methods convert between common programming naming conventions:

Listing 8.12: Naming convention transformations

```
let identifier = "getUserName"

print(identifier.snake_case())  // Output: get_user_name
print(identifier.kebab_case())  // Output: get-user-name
print(identifier.camel_case())  // Output: getUserName
print(identifier.title_case())  // Output: Getusername
```

The transformations handle various input formats:

Listing 8.13: Cross-format conversion

```
// From snake_case
print("user_full_name".camel_case())  // Output: userFullName
print("user_full_name".kebab_case())  // Output: user-full-name

// From kebab-case
print("my-component-name".snake_case()) // Output: my_component_name
print("my-component-name".camel_case()) // Output: myComponentName

// capitalize: first letter uppercase, rest lowercase
print("hello world".capitalize()) // Output: Hello world
```

Three of these four—`snake_case`, `kebab_case`, `camel_case`—share one notion of a word boundary, implemented in `is_case_boundary` in `src/string_ops.c`: a space, a hyphen, an underscore, or a lowercase letter immediately followed by an uppercase one. That last clause is what lets `"getUserName"` decompose. It is also what makes acronyms disappear: in `"XMLParser"` every adjacent pair is upper-upper or upper-lower, so no boundary is ever detected and `"XMLParser".snake_case()` produces the run-together `"xmlparser"`.

`title_case` is the odd one out. It uppercases the first letter of each run and lowercases the rest, but it treats *only* whitespace, _, and - as separators, and it copies those separators through unchanged. It is a title-caser for prose, not a converter between naming conventions:

Listing 8.14: title_case keeps its separators

```
print("hello world".title_case())    // Output: Hello World
print("hello_world".title_case())    // Output: Hello_World
print("user_full_name".title_case()) // Output: User_Full_Name
print("getUserName".title_case())    // Output: Getusername
```

To turn `"user_full_name"` into the UI label `"User Full Name"`, replace the separators first: `"user_full_name".replace("_", " ").title_case()`.

When to Use Case Transformations

Case transformations earn their keep in code generation, template engines, and API adapters. A database column named `"first_name"` becomes `"firstName"` for a JSON API with `.camel_case()`, and `"First Name"` for a UI label with `.replace("_", " ").title_case()`.

Method	Input	Output
.to_upper()	"hello"	"HELLO"
.to_lower()	"HELLO"	"hello"
.capitalize()	"hello world"	"Hello world"
.snake_case()	"getUserName"	"get_user_name"
.camel_case()	"get_user_name"	"getUserName"
.kebab_case()	"getUserName"	"get-user-name"
.title_case()	"hello world"	"Hello World"

Table 8.1: Case transformation methods

8.4 Characters, Bytes, and Substrings

8.4.1 chars() and bytes()

The chars method returns an array of one-byte strings, while bytes returns the same sequence as integers:

Listing 8.15: chars and bytes

```
let word = "Hello"

print(word.chars()) // Output: [H, e, l, l, o]
print(word.bytes()) // Output: [72, 101, 108, 108, 111]
```

chars() Splits Bytes, Not Code Points

For ASCII text the two readings coincide, which is why chars feels character-shaped. It is not. Put a single Greek lambda (λ) in a string literal and len() answers 2, because that is two UTF-8 bytes—bytes() returns [206, 187], which is 0xCE 0xBB—and chars() returns *two* one-byte strings, neither of which is a valid character on its own. Reversing or indexing a string containing non-ASCII text will therefore scramble it. If you need code points, decode the UTF-8 yourself from bytes().

The chars method is useful for character-level processing:

Listing 8.16: Character-level operations

```
fn is_palindrome(s: String) -> Bool {
  let chars = s.to_lower().chars()
  let n = chars.len()
  for i in 0..(n / 2) {
    if chars[i] != chars[n - 1 - i] {
      return false
    }
  }
  true
}

print(is_palindrome("racecar")) // Output: true
print(is_palindrome("hello"))   // Output: false
print(is_palindrome("Madam"))   // Output: true
```

8.4.2 len(), substring(), and index_of()

Listing 8.17: String length, substrings, and searching

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

// Length
print(message.len()) // Output: 13

// Substring extraction (start inclusive, end exclusive)
print(message.substring(0, 5))  // Output: Hello
print(message.substring(7, 12)) // Output: World

// Finding positions
print(message.index_of("World"))  // Output: 7
print(message.index_of("world"))  // Output: -1 (case-sensitive)

// Last occurrence
let repeated = "abcabc"
print(repeated.last_index_of("abc")) // Output: 3
print(repeated.index_of("abc"))      // Output: 0
```

`last_index_of` arrived in v0.3.29 alongside the three predicates we'll meet in a moment; before that, finding the final occurrence meant scanning by hand.

The `substring` method supports negative indices, which count from the end of the string, and clamps out-of-bounds values instead of raising:

Listing 8.18: Negative indices in substring

```
let text = "Hello, World!"

print(text.substring(-6, -1)) // Output: World
```

Negative Substring Bounds Need a VM

The default stack VM and `--regvm` both read `substring(-6, -1)` as "six from the end, up to one from the end." The tree-walking interpreter clamps the negatives to zero instead and hands back the empty string. Positive bounds behave identically everywhere, so `text.substring(7, 12)` is the portable spelling.

8.4.3 More String Utilities

Listing 8.19: reverse, repeat, and padding

```
// Reverse
print("desserts".reverse()) // Output: stressed

// Repeat
print("ha".repeat(3))    // Output: hahaha
print("-".repeat(20))    // Output: --------------------

// Padding: both arguments are required
print("[" + "42".pad_left(5, " ") + "]")   // Output: [   42]
print("[" + "42".pad_left(5, "0") + "]")   // Output: [00042]
print("[" + "hi".pad_right(10, ".") + "]") // Output: [hi........]
```

pad_left Takes Two Arguments

The pad character is not optional on the default VM or the tree-walker: `"42".pad_left(5)` raises `type 'String' has no method 'pad_left'` on the stack VM and `.pad_left() expects 2 arguments (n, ch)` under `--tree-walk`. (The register backend does default it to a space—don't rely on that.) Pass the pad character explicitly and the line works everywhere.

`repeat` has one edge worth knowing: when the requested total would overflow, it does not raise—it returns the empty string. `"xy".repeat(9223372036854775807).len()` is `0` on the default VM and on `--tree-walk`, as are `"xy".repeat(0)` and `"".repeat(5)`. Guard the count yourself if a zero-length result would be a silent bug. (On `--regvm` the same overflowing call reports an error instead of returning `""`; either way you do not get a two-exabyte string.)

Four predicates round out the inspection vocabulary. `is_alpha`, `is_digit`, and `is_alphanumeric` joined `is_empty` in v0.3.29:

Listing 8.20: String validation methods

```
print("hello".is_alpha())          // Output: true
print("hello123".is_alpha())       // Output: false
print("hello world".is_alpha())    // Output: false  -- the space disqualifies it
print("12345".is_digit())          // Output: true
print("abc123".is_alphanumeric()) // Output: true
print("".is_empty())               // Output: true
print("".is_alpha())               // Output: false  -- so does emptiness
print("hi".is_empty())             // Output: false
```

All three predicates demand that *every* byte qualifies, and all three answer `false` for the empty string.

8.5 Binary-Safe Strings

Most languages that grew up around C inherited a quiet assumption: a string ends at the first zero byte. Lattice used to inherit it too. As of v0.6.4 it does not.

Binary-Safe String

A string is *binary-safe* when its length, and not a terminator, decides where it ends. A binary-safe string may contain any byte value—including `0x00`—and every operation on it must consult the stored length rather than scanning for a NUL.

Method	Returns	Description
`.len()`	`Int`	String length in bytes
`.chars()`	`Array`	Array of characters
`.bytes()`	`Array`	Array of byte values
`.substring(s, e)`	`String`	Extract range `[s, e)`
`.index_of(sub)`	`Int`	First position, or `-1`
`.last_index_of(sub)`	`Int`	Last position, or `-1`
`.reverse()`	`String`	Characters in reverse
`.repeat(n)`	`String`	Repeated `n` times
`.pad_left(w, ch)`	`String`	Left-pad to width
`.pad_right(w, ch)`	`String`	Right-pad to width
`.is_empty()`	`Bool`	Is the string empty?
`.is_alpha()`	`Bool`	All alphabetic?
`.is_digit()`	`Bool`	All digits?
`.is_alphanumeric()`	`Bool`	All alphanumeric?

Table 8.2: Character and substring methods

This is not an exotic corner. Read a PNG header, decode a length-prefixed wire protocol, capture the stdout of a program that emits packed records, parse JSON containing a `\u0000` escape—and a NUL lands in a Lattice string. Before v0.6.4 the string silently ended there. Now it doesn't.

8.5.1 Getting a NUL Into a String

The one route that does *not* work is the source file. As we saw earlier, there is no `\0` escape, and `\x00` truncates the literal too. Nor will `chr(0)` help: it returns the empty string.

The dependable route is a `Buffer` (Section 22.5), which is a byte array with no opinions at all:

Listing 8.21: Building a string with an embedded NUL

```
flux packet = Buffer::new(3)
packet.write_u8(0, 65)   // 'A'
packet.write_u8(1, 0)    // NUL
packet.write_u8(2, 66)   // 'B'

let framed = packet.to_string()
print(framed.len())      // Output: 3
print(framed.bytes())    // Output: [65, 0, 66]
```

Three bytes in, three bytes out. `print(framed)` writes all three to standard output, NUL included—your terminal will show something like `A^@B` if you pipe it through `cat -v`.

The same bytes arrive from `read_file_bytes`, from socket reads, from `exec_argv` output, and from `json_parse`. `Buffer` is the one route we can spell in a listing.

8.5.2 Every Method Honors the Length

Here is the part that matters. Searching, splitting, slicing, concatenating, interpolating, reversing, case-folding, and round-tripping back through a `Buffer` all count bytes:

Listing 8.22: String methods on NUL-bearing data

```
fn nul_byte() -> String {
  flux b = Buffer::new(1)
  b.write_u8(0, 0)
  b.to_string()
}

let nul = nul_byte()

print(framed.contains(nul))              // Output: true
print(framed.index_of(nul))              // Output: 1
print(framed.last_index_of(nul))         // Output: 1
print(framed.split(nul).len())           // Output: 2
print(framed.split(nul))                 // Output: [A, B]
print(framed.substring(0, 1))            // Output: A
print(framed.reverse().bytes())          // Output: [66, 0, 65]
print(framed.to_lower().bytes())         // Output: [97, 0, 98]
print((framed + framed).len())           // Output: 6
print("${framed}|".len())                // Output: 4
print(Buffer::from_string(framed).len()) // Output: 3
```

Read that list once more with C-string eyes and every line would have answered differently: `len` would be 1, `split` would find nothing, concatenation would produce 2 bytes, and the `Buffer` round-trip would lose two thirds of the payload.

Comparison and ordering are byte-exact too, which means sorting and equality see past the NUL rather than stopping at it:

Listing 8.23: Equality, ordering, and format

```
fn framed_with(last: Int) -> String {
  flux b = Buffer::new(3)
  b.write_u8(0, 65)
  b.write_u8(1, 0)
  b.write_u8(2, last)
  b.to_string()
}

let ab = framed_with(66)   // 'A', NUL, 'B'
let ac = framed_with(67)   // 'A', NUL, 'C'

print(ab == ac)                 // Output: false
print(ab == framed_with(66))    // Output: true

let ordered = [ac, ab].sort()
print(ordered[0].bytes())       // Output: [65, 0, 66]
print(ordered[1].bytes())       // Output: [65, 0, 67]

print(format("[{}]", ab).len()) // Output: 5
```

Equality compares lengths and then the bytes; `sort` orders on the full byte sequence, so `ab` really does sort before `ac` on the strength of a difference that lives *after* a NUL. All of this behaves identically on the default stack VM, on `--tree-walk`, and on `--regvm`.

8.5.3 Where the Old Assumption Survives

A handful of surfaces still hand the bytes to C library routines, and those routines still stop at the first zero. These are the ones to know:

Listing 8.24: Surfaces that still truncate

```
flux index = Map::new()
index[framed] = 1
print(index.keys()[0].len())      // Output: 1  -- map key truncated

print(framed.capitalize().len())  // Output: 1
print(framed.snake_case().len())  // Output: 1
print(framed.title_case().len())  // Output: 1
```

Do Not Use Binary Data as a Map Key

Map keys are stored as C strings, so a key containing a NUL is silently cut short—two different payloads can collide into the same entry. The case transformers (`capitalize`, `title_case`, `snake_case`, `camel_case`, `kebab_case`) truncate for the same reason, as do `String` payloads attached to raised errors. Key your maps on something textual—`to_hex()` of the corresponding `Buffer` is a reasonable choice—and keep the raw bytes in the value.

`Buffer::new` Wants Its Size

As of vo.6.4, `Buffer::new()` with no argument (or a non-`Int` one) raises `Buffer::new() expects 1 Int argument` on all three backends. Earlier releases handed back an empty buffer on the VMs and left you wondering why nothing was written. Reads past the end are equally explicit—`Buffer::new(4).read_u64(0)` raises `Buffer.read_u64: index out of bounds`, following the pattern `Buffer.<method>: index out of bounds`.

8.6 Character Codes: ord and chr

Lattice provides two built-in functions for working with character codes: `ord()` and `chr()`.

8.6.1 ord() and chr()

The `ord` function returns the numeric code of a character, and `chr` returns the character for a given code:

Listing 8.25: Converting between characters and codes

```
print(ord("A"))  // Output: 65
print(ord("a"))  // Output: 97
print(ord("0"))  // Output: 48
print(ord("\n")) // Output: 10

print(chr(65))  // Output: A
print(chr(97))  // Output: a
print(chr(48))  // Output: 0
```

ord() and chr()

`ord(s)` returns the code point of the first character in string `s`. If the string is empty, it returns `-1`. `chr(n)` returns a one-character string for the code point `n` (currently limited to 0–127, the ASCII range).

These functions are useful for character classification, encoding, and building custom parsers:

Listing 8.26: Character classification with ord

```
fn is_uppercase(ch: String) -> Bool {
  let code = ord(ch)
  code >= 65 && code <= 90
}

fn is_lowercase(ch: String) -> Bool {
  let code = ord(ch)
  code >= 97 && code <= 122
}

fn rot13(text: String) -> String {
  let chars = text.chars()
  flux result = ""
  for ch in chars {
    let code = ord(ch)
    if is_uppercase(ch) {
      result = result + chr((code - 65 + 13) % 26 + 65)
    } else if is_lowercase(ch) {
      result = result + chr((code - 97 + 13) % 26 + 97)
    } else {
      result = result + ch
    }
  }
  result
}

print(rot13("Hello, World!")) // Output: Uryyb, Jbeyq!
print(rot13("Uryyb, Jbeyq!")) // Output: Hello, World!
```

8.6.2 Building a Caesar Cipher

Listing 8.27: A simple Caesar cipher

```
fn caesar_encrypt(text: String, shift: Int) -> String {
  flux encrypted = ""
  for ch in text.chars() {
    let code = ord(ch)
    if code >= 65 && code <= 90 {
      // Uppercase letter
      encrypted = encrypted + chr((code - 65 + shift) % 26 + 65)
    } else if code >= 97 && code <= 122 {
      // Lowercase letter
      encrypted = encrypted + chr((code - 97 + shift) % 26 + 97)
    } else {
      encrypted = encrypted + ch
    }
  }
  encrypted
}

fn caesar_decrypt(text: String, shift: Int) -> String {
  caesar_encrypt(text, 26 - shift)
}

let secret = caesar_encrypt("Attack at dawn", 3)
print(secret)                       // Output: Dwwdfn dw gdzq
print(caesar_decrypt(secret, 3))    // Output: Attack at dawn
```

8.7 Regular Expressions

Lattice includes built-in regular expression support via three *free functions*: `regex_match`, `regex_find_all`, and `regex_replace`. These use POSIX extended regular expressions under the hood (implemented in `src/regex_ops.c`).

Functions, Not Methods—and the Pattern Comes First

These three are not string methods. `"abc".regex_match("a")` fails on every backend—`type 'String' has no method 'regex_match'` on the default VM, `unknown method '.regex_match()' on String` under `--tree-walk`, `no method 'regex_match' on String` under `--regvm`.
Call them as functions, and note the argument order: the *pattern* is first, the subject string second. That is the opposite of what most languages with a method form condition you to expect. Section 25.6 documents the same three functions from the standard-library side.

8.7.1 regex_match — Test a Pattern

`regex_match(pattern, s)` tests whether a string matches a regular expression pattern:

Listing 8.28: Pattern matching with regex

```
let email = "alice@example.com"

print(regex_match("[a-zA-Z]+@[a-zA-Z]+\\.[a-zA-Z]+", email))
// Output: true

print(regex_match("[a-zA-Z]+@[a-zA-Z]+\\.[a-zA-Z]+", "not-an-email"))
// Output: false

// Validate a phone number format
let phone = "555-123-4567"
print(regex_match("[0-9]{3}-[0-9]{3}-[0-9]{4}", phone))
// Output: true
```

8.7.2 regex_find_all — Extract All Matches

`regex_find_all(pattern, s)` returns an array of all substrings matching the pattern:

Listing 8.29: Finding all regex matches

```
let text = "Call 555-1234 or 555-5678 for info"

let phone_numbers = regex_find_all("[0-9]{3}-[0-9]{4}", text)
print(phone_numbers) // Output: [555-1234, 555-5678]

// Extract all words
let words = regex_find_all("[a-zA-Z]+", "Hello, World! How are you?")
print(words) // Output: [Hello, World, How, are, you]

// Find all numbers (including decimals)
let data = "Price: $12.50, Tax: $1.00, Total: $13.50"
let amounts = regex_find_all("[0-9]+\\.[0-9]+", data)
print(amounts) // Output: [12.50, 1.00, 13.50]
```

Note that `print` renders an array of strings without quotation marks around the elements, so the output reads `[555-1234, 555-5678]` rather than `["555-1234", "555-5678"]`.

8.7.3 regex_replace — Pattern-Based Replacement

`regex_replace(pattern, s, replacement)` replaces all matches of a pattern with a replacement string. The subject sits in the middle:

Listing 8.30: Regex replacement

```
// Normalize whitespace
let messy = "too   many    spaces   here"
let clean = regex_replace("[ ]+", messy, " ")
print(clean) // Output: too many spaces here

// Remove all non-alphanumeric characters
let raw = "Hello, World! (2024)"
let cleaned = regex_replace("[^a-zA-Z0-9 ]", raw, "")
print(cleaned) // Output: Hello World 2024

// Redact phone numbers
let log = "User called from 555-1234, callback to 555-5678"
let redacted = regex_replace("[0-9]{3}-[0-9]{4}", log, "XXX-XXXX")
print(redacted) // Output: User called from XXX-XXXX, callback to XXX-XXXX
```

8.7.4 Regex Flags

All three regex functions accept an optional flags string as the last argument:

Listing 8.31: Case-insensitive regex

```
let text = "Hello HELLO hello HeLLo"

// Case-insensitive matching
let matches = regex_find_all("hello", text, "i")
print(matches) // Output: [Hello, HELLO, hello, HeLLo]

// Case-insensitive replacement
let normalized = regex_replace("hello", text, "hi", "i")
print(normalized) // Output: hi hi hi hi
```

The available flags are:

- `"i"`: Case-insensitive matching
- `"m"`: Multiline mode (newlines treated as line boundaries)

POSIX Regex Syntax

Lattice uses POSIX extended regular expressions, not Perl-compatible (PCRE) syntax. This means some patterns you may be used to from other languages work differently:

- Use `[0-9]` instead of `\\d`
- Use `[a-zA-Z0-9_]` instead of `\\w`
- Lookaheads and lookbehinds are not supported

8.8 Putting It All Together

Let's combine these string methods into a practical example—a log file analyzer:

Listing 8.32: A log file analyzer

```
fn analyze_log(log_text: String) -> Map {
  let lines = log_text.split("\n").filter(|l| !l.is_empty())
  let results = Map::new()

  // Count log levels
  let levels = Map::new()
  levels["INFO"] = 0
  levels["WARN"] = 0
  levels["ERROR"] = 0

  for line in lines {
    if line.contains("INFO") {
      levels["INFO"] = levels["INFO"] + 1
    } else if line.contains("WARN") {
      levels["WARN"] = levels["WARN"] + 1
    } else if line.contains("ERROR") {
      levels["ERROR"] = levels["ERROR"] + 1
    }
  }
  results["levels"] = levels
  results["total_lines"] = lines.len()

  // Extract timestamps
  let timestamps = []
  for line in lines {
    let ts_matches = regex_find_all("[0-9]{4}-[0-9]{2}-[0-9]{2}", line)
    for ts in ts_matches {
      timestamps.push(ts)
    }
  }
  results["dates"] = timestamps.unique()

  // Extract error messages
  let errors = lines
    .filter(|l| l.contains("ERROR"))
    .map(|l| {
      let parts = l.split("ERROR")
      if parts.len() > 1 {
        parts[1].trim()
      } else {
        l
      }
    })
  results["errors"] = errors

  results
}
```

8.9 Exercises

1. **String Compression.** Write a function that performs basic run-length encoding: `"aaabbbcc"` becomes `"a3b3c2"`. Handle single characters (no count needed for runs of 1).
2. **CSV Parser.** Write a function that takes a CSV string (with header row) and returns an array of maps, where each map represents a row with column names as keys. Handle quoted fields if possible.
3. **Slug Generator.** Write a function that converts a title like `"Hello, World! This is Great"` into a URL slug like `"hello-world-this-is-great"`. Use `to_lower`, `regex_replace`, and `trim`.
4. **Password Validator.** Write a function that checks if a password meets these criteria: at least 8 characters, contains at least one uppercase letter, one lowercase letter, one digit, and one special character. Use `regex_match` or character inspection.
5. **Template Engine.** Write a simple template engine where `render("Hello \{\{name\}\}, age \{\{age\}\}", context)` replaces `\{\{name\}\}` and `\{\{age\}\}` with values from a context map. Use `regex_find_all` to find placeholders and `replace` to fill them in.

What's Next?

Strings give us the ability to represent and manipulate text, but programs often need to inspect the *structure* of data—not just its content. In Chapter 9, we'll explore Lattice's `match` expression, which lets you destructure arrays, structs, and enums, test ranges, and make decisions based on the shape of your data. Pattern matching is one of Lattice's most powerful features, and it builds naturally on everything you've learned so far.

Chapter 9

Pattern Matching

Every program makes decisions. In Chapter 5 we used `if`/`else` chains to steer execution based on boolean conditions, and that works beautifully when you have one or two tests to run. But programs often face a harder question: "What *shape* does this data have?" Is the HTTP response a success or a failure? Does the command-line argument look like a number, a flag, or a filename? Is the list empty, a singleton, or something longer?

Pattern matching answers these questions directly. Instead of writing a cascade of `if`/`else` branches—each one manually testing a field or comparing a value—you describe what the data *looks like*, and Lattice picks the first description that fits.

Listing 9.1: A first taste of match

```
let status_code = 404

let message = match status_code {
    200 => "OK",
    301 => "Moved Permanently",
    404 => "Not Found",
    500 => "Internal Server Error",
    _   => "Unknown status"
}

print(message) // Not Found
```

This chapter builds from that starting point. We will explore every pattern type Lattice supports—from literals and wildcards through range patterns, array destructuring, struct field extraction, and

enum variant matching. Along the way we will meet *guards*, *bindings*, and *exhaustiveness checking*, the compiler's way of telling you when you have forgotten a case.

9.1 The `match` Expression

A `match` expression evaluates a *scrutinee* (the value being inspected) and compares it against a series of *arms*. Each arm has a pattern on the left side of `=>` and a body on the right:

Match Expression

```
match scrutinee {
    pattern1 => body1,
    pattern2 => body2,
    _        => fallback_body
}
```

A *match expression* evaluates the scrutinee once, then tries each arm's pattern from top to bottom. The body of the first matching arm is executed, and its value becomes the value of the entire `match` expression. If no arm matches, the expression evaluates to `nil`.

Because `match` is an expression, you can bind its result to a variable, return it from a function, or embed it anywhere a value is expected:

Listing 9.2: match as an expression

```
fn describe_temperature(temp: Int) -> String {
    return match temp {
        0   => "Freezing point of water",
        100 => "Boiling point of water",
        _   => "Just another temperature: ${temp}"
    }
}

print(describe_temperature(100)) // Boiling point of water
print(describe_temperature(42))  // Just another temperature: 42
```

9.1.1 Arm Bodies: Expressions and Blocks

An arm body can be a single expression (as we have seen) or a block enclosed in curly braces. When a block is used, the value of the last expression in the block becomes the arm's value:

Listing 9.3: Block bodies in match arms

```
let score = 87

let grade = match score {
    100 => "Perfect!",
    _ => {
        let letter = if score >= 90 { "A" } else { "B" }
        "${letter} (${score}/100)"
    }
}

print(grade) // B (87/100)
```

Commas Between Arms

Arms are separated by commas. The trailing comma after the last arm is optional but encouraged for consistency—it makes adding new arms later easier and produces cleaner diffs in version control.

9.1.2 First Match Wins

Arms are tried from top to bottom, and the first one that matches wins. This ordering matters when patterns overlap:

Listing 9.4: Order matters—first match wins

```
let n = 0

let result = match n {
    0 => "zero",       // This arm matches first
    _ => "something",  // This arm would also match, but never reached
}

print(result) // zero
```

If you put a wildcard _ before more specific patterns, the specific patterns will never be reached. Lattice currently does not warn about unreachable arms, so you should arrange your patterns from most specific to least specific.

No Match Returns nil

If no arm matches the scrutinee, the entire `match` expression evaluates to `nil`—not an error. This is a deliberate design choice that keeps programs running, but it also means a missing arm might silently produce `nil` where you expected a real value. The exhaustiveness checker (Section 9.9) helps catch these situations at compile time.

9.2 Literal Patterns

The most direct kind of pattern matches a specific, concrete value. Literal patterns support integers, floats, strings, booleans, and `nil`:

Listing 9.5: Literal patterns across types

```
fn classify(value: any) -> String {
    return match value {
        42      => "The answer",
        3.14    => "Approximately pi",
        "hello" => "A greeting",
        true    => "Affirmative",
        nil     => "Nothing at all",
        _       => "Something else"
    }
}

print(classify(42))      // The answer
print(classify("hello")) // A greeting
print(classify(nil))     // Nothing at all
print(classify(99))      // Something else
```

Literal patterns use value equality to compare. Two integers are equal if they hold the same number; two strings are equal if they contain the same characters. Complex types like arrays and structs are *not* compared by value in literal patterns—use destructuring patterns (Sections 9.6 and 9.7) for those.

Negative numeric literals work as you would expect:

Listing 9.6: Negative literal patterns

```
let temp = -40

let note = match temp {
    -40 => "Where Celsius meets Fahrenheit",
    0   => "Freezing",
    _   => "Unremarkable"
}

print(note) // Where Celsius meets Fahrenheit
```

9.3 Wildcard and Binding Patterns

9.3.1 The Wildcard _

The underscore _ matches any value without capturing it. It is the catch-all, the "I don't care what this is" pattern:

Listing 9.7: Wildcard as a catch-all

```
fn is_special(n: Int) -> Bool {
    return match n {
        0 => true,
        1 => true,
        _ => false
    }
}
```

A wildcard arm placed at the end of a `match` guarantees that every possible value is covered, which makes the match exhaustive.

9.3.2 Binding Patterns

Sometimes you want to match *any* value, but you also want to use it inside the arm's body. A binding pattern does exactly that—any bare identifier (other than _) becomes a variable bound to the scrutinee's value:

Listing 9.8: Binding patterns capture the scrutinee

```
let response_code = 418

let msg = match response_code {
    200 => "Success",
    404 => "Not found",
    code => "Unexpected status: ${code}"
}

print(msg) // Unexpected status: 418
```

The name `code` in that last arm is not a keyword or a previously defined variable—it is a fresh local variable whose value is the scrutinee (here, `418`). The binding is only available inside the arm's body.

Bindings vs. Literals

How does Lattice tell the difference between a binding pattern like `code` and a literal pattern? The rule is: literal patterns are actual literal values (`42`, `"hello"`, `true`, `nil`). Any other bare identifier is treated as a binding. You cannot match against the value of an existing variable by naming it in a pattern—use a guard instead (Section 9.5).

Binding patterns truly shine when combined with guards, which we will explore in Section 9.5.

9.4 Range Patterns

Range patterns match values that fall within a numeric range. The syntax uses the familiar `..` operator, but with an important twist: range patterns are **inclusive** on both ends.

Listing 9.9: Grading with range patterns

```
fn letter_grade(score: Int) -> String {
    return match score {
        90..100 => "A",
        80..89  => "B",
        70..79  => "C",
        60..69  => "D",
        0..59   => "F",
        _       => "Invalid score"
    }
}

print(letter_grade(95))  // A
print(letter_grade(80))  // B
print(letter_grade(70))  // C
print(letter_grade(42))  // F
```

Ranges in Patterns vs. Ranges Elsewhere

The `..` operator has different semantics depending on context. When used to create a `Range` value (for example, in a `for` loop), `a..b` is *half-open*: it includes `a` but excludes `b`. Inside a `match` pattern, however, `a..b` is *inclusive*: it matches values where $a \leq \text{value} \leq b$. This is a deliberate choice—inclusive ranges make match arms more intuitive when partitioning a numeric domain into adjacent regions.

Under the hood, the compiler translates a range pattern into a pair of comparisons. The pattern `80..89` becomes roughly `value >= 80 && value <= 89`, using `OP_GTEQ` and `OP_LTEQ` opcodes connected by a short-circuit jump. You can see this in the range-pattern compilation logic in `src/stackcompiler.c`.

Range patterns work with integer values. You can also use variables or expressions as the endpoints:

Listing 9.10: Dynamic range endpoints

```
let threshold = 50

let category = match 42 {
    0..threshold => "Below or at threshold",
    _            => "Above threshold"
}

print(category) // Below or at threshold
```

9.5 Guards

A *guard* is an additional boolean condition attached to a match arm with the `if` keyword. The arm only matches if both the pattern and the guard are satisfied:

Listing 9.11: Guards refine pattern matches

```
fn classify_number(n: Int) -> String {
    return match n {
        0 => "zero",
        x if x > 0 => "positive: ${x}",
        x if x < 0 => "negative: ${x}",
        _ => "unreachable"
    }
}

print(classify_number(42))  // positive: 42
print(classify_number(-7))  // negative: -7
print(classify_number(0))   // zero
```

In the arms `x if x > 0` and `x if x < 0`, the identifier `x` is a binding pattern that captures the scrutinee, and the `if` clause is a guard that tests the captured value. The guard expression can use any bound variables from the pattern, making it possible to write powerful, flexible conditions.

Guards are full expressions, so you can call functions, combine boolean operators, or access methods:

Listing 9.12: Complex guard expressions

```
fn describe_name(name: String) -> String {
    return match name {
        s if s.len() == 0    => "Empty name",
        s if s.len() > 50    => "That's a very long name",
        s if s.contains(" ") => "Full name: ${s}",
        s                    => "First name only: ${s}"
    }
}

print(describe_name(""))                // Empty name
print(describe_name("Ada"))             // First name only: Ada
print(describe_name("Ada Lovelace"))    // Full name: Ada Lovelace
```

Guards and Exhaustiveness

A guarded arm is not considered a catch-all by the exhaustiveness checker, even if its pattern is a wildcard or a binding. The guard might evaluate to `false`, so the checker cannot guarantee the arm will match. To ensure full coverage, add an unguarded catch-all arm at the end.

9.5.1 Guards with Other Pattern Types

Guards can be combined with any pattern type. Here is a range pattern with a guard:

Listing 9.13: Combining range patterns and guards

```
fn shipping_message(weight: Int) -> String {
    return match weight {
        1..5 if weight == 1   => "Letter rate",
        1..5                  => "Small parcel",
        6..30                 => "Standard parcel",
        _ if weight > 1000    => "Freight only",
        _                     => "Large parcel"
    }
}

print(shipping_message(1))    // Letter rate
print(shipping_message(3))    // Small parcel
print(shipping_message(2000)) // Freight only
```

9.6 Array Destructuring Patterns

So far, every pattern has matched a single flat value. Array patterns reach *inside* a value, matching the structure and extracting individual elements:

Listing 9.14: Basic array destructuring

```
let point = [10, 20]

match point {
    [0, 0]    => print("Origin"),
    [x, 0]    => print("On the x-axis at ${x}"),
    [0, y]    => print("On the y-axis at ${y}"),
    [x, y]    => print("Point at (${x}, ${y})")
}
// Point at (10, 20)
```

An array pattern is enclosed in square brackets and contains sub-patterns for each element. The scrutinee must be an array, and—without a rest pattern—its length must exactly match the number of elements in the pattern.

A Warning You Will See

Run that listing and, alongside the output, you will see this on standard error:

```
warning: non-exhaustive match on line 3: consider adding a wildcard `_` arm
```

The exhaustiveness checker cannot prove that four array patterns cover every possible array, so it asks for a wildcard. The warning goes to *stderr*, never to stdout, and it never stops the program—which is why the listings in this chapter show only the values they print. Section 9.9 explains exactly which matches the checker can and cannot reason about.

Each element position can hold any sub-pattern:

- A **literal** like `0` matches that specific value at that position.
- A **binding** like `x` matches any value and captures it.
- A **wildcard** `_` matches any value without capturing.

Listing 9.15: Mixing literal and binding sub-patterns

```
let rgb = [255, 128, 0]

let color_name = match rgb {
    [255, 0, 0]     => "Red",
    [0, 255, 0]     => "Green",
    [0, 0, 255]     => "Blue",
    [255, 255, 255] => "White",
    [0, 0, 0]       => "Black",
    [r, g, b]       => "Custom RGB(${r}, ${g}, ${b})"
}

print(color_name) // Custom RGB(255, 128, 0)
```

9.6.1 Rest Patterns

What if you do not know (or care about) how many elements the array has? The rest pattern `...name` collects zero or more remaining elements into a sub-array:

Listing 9.16: Rest patterns collect remaining elements

```
let numbers = [1, 2, 3, 4, 5]

match numbers {
    []           => print("Empty"),
    [only]       => print("Just one: ${only}"),
    [first, ...rest] => {
        print("First: ${first}")
        print("Rest: ${rest}")
    }
}
// First: 1
// Rest: [2, 3, 4, 5]
```

The rest pattern is powerful for processing lists head-first. You can place elements *after* the rest pattern too, matching from both ends:

Listing 9.17: Rest with trailing elements

```
let log_entries = ["INFO", "Starting up", "v2.1", "2024-01-15"]

match log_entries {
    [level, ..middle, date] => {
        print("Level: ${level}")
        print("Date: ${date}")
        print("Middle: ${middle}")
    },
    _ => print("Unexpected format")
}
// Level: INFO
// Date: 2024-01-15
// Middle: [Starting up, v2.1]
```

The compiler handles post-rest elements by indexing from the end of the array. For a pattern like `[first, ..mid, last]`, the element `last` is extracted at index `array.len() - 1`, and `mid` receives a slice of everything in between.

Length Requirements with Rest

Without a rest pattern, the array length must match the pattern count exactly. With a rest pattern, the array must have *at least* as many elements as the non-rest positions. A pattern `[a, ..rest, b]` requires at least two elements; the rest can be empty.

9.6.2 Array Patterns and Guards

Array patterns combine naturally with guards for precise conditions:

Listing 9.18: Array patterns with guards

```
fn describe_pair(pair: Array) -> String {
    return match pair {
        [a, b] if a == b     => "Equal pair: ${a}",
        [a, b] if a + b == 0 => "Opposite pair: ${a} and ${b}",
        [a, b]               => "Pair: ${a} and ${b}",
        _                    => "Not a pair"
    }
}

print(describe_pair([5, 5]))    // Equal pair: 5
print(describe_pair([3, -3]))   // Opposite pair: 3 and -3
print(describe_pair([1, 2]))    // Pair: 1 and 2
```

9.7 Struct Destructuring Patterns

Struct patterns match values by their field names and optionally by field values. They use curly braces and are especially useful for pulling out the fields you need without cluttering your code with manual field access.

Listing 9.19: Basic struct destructuring

```
struct User {
    name: String,
    age: Int,
    active: Bool
}

let user = User { name: "Rosalind", age: 33, active: true }

match user {
    {active: false} => print("Inactive user"),
    {name, age}     => print("${name}, age ${age}")
}
// Rosalind, age 33
```

There are two forms of struct field patterns:

1. **Shorthand binding**: Writing just a field name like `{name, age}` matches any value for those fields and binds them to variables of the same name.

2. **Value matching**: Writing `{active: false}` matches only when the `active` field equals `false`.

You can mix both forms in a single pattern:

Listing 9.20: Mixed shorthand and value patterns

```
struct Config {
    mode: String,
    debug: Bool,
    port: Int
}

let config = Config { mode: "production", debug: false, port: 8080 }

match config {
    {mode: "test", debug: true} => print("Test mode with debugging"),
    {mode: "production", port}  => print("Production on port ${port}"),
    {mode}                      => print("Running in ${mode} mode")
}
// Production on port 8080
```

Partial Matching

Struct patterns do not need to mention every field. A pattern like `\{name\}` matches any struct that has a `name` field, regardless of what other fields it contains or what their values are. This makes struct patterns resilient to additions—if you add a field to the struct later, existing patterns continue to work.

9.8 Enum Variant Patterns

Enum variant patterns are where pattern matching truly shines. They let you distinguish between an enum's variants and extract the data each variant carries:

Listing 9.21: Matching enum variants

```
enum Shape {
    Circle(Float),
    Rectangle(Float, Float),
    Triangle(Float, Float, Float)
}

fn area(shape: Shape) -> Float {
    return match shape {
        Shape::Circle(r)           => 3.14159 * r * r,
        Shape::Rectangle(w, h)     => w * h,
        Shape::Triangle(a, b, c)   => {
            let s = (a + b + c) / 2.0
            // Heron's formula
            let val = s * (s - a) * (s - b) * (s - c)
            // Approximate square root via Newton's method
            flux guess = val / 2.0
            for _ in 0..10 {
                guess = (guess + val / guess) / 2.0
            }
            guess
        }
    }
}

print(area(Shape::Circle(5.0)))          // 78.5397
print(area(Shape::Rectangle(3.0, 4.0)))  // 12
```

The pattern `Shape::Circle(r)` checks three things: (1) the value is an enum, (2) it belongs to the `Shape` enum, and (3) its variant is `Circle`. If all three pass, the payload is bound to `r`.

Payload Destructuring Arrived in v0.3.29

Destructuring an enum payload directly in the pattern is the way to do this, and it has been since v0.3.29. Older Lattice code sometimes matches the bare variant and then digs the payload out by hand—calling `.payload()` and indexing into the resulting array. If you inherit code shaped like that, collapse it into a payload pattern; the compiler emits the same extraction for you, and the binding is scoped to the arm.

9.8.1 Variants Without Payloads

For unit variants (those carrying no data), use the variant name without parentheses:

Listing 9.22: Matching unit variants

```
enum Direction {
    North,
    South,
    East,
    West
}

fn arrow(dir: Direction) -> String {
    return match dir {
        Direction::North => "^",
        Direction::South => "v",
        Direction::East  => ">",
        Direction::West  => "<"
    }
}

print(arrow(Direction::North)) // ^
print(arrow(Direction::West))  // <
```

9.8.2 Nested Enum Patterns

Each payload position in an enum variant pattern can itself be a sub-pattern—a literal to match a specific value, a binding to capture the value, or a wildcard to ignore it:

Listing 9.23: Sub-patterns in enum payloads

```
enum Result {
    Ok(any),
    Err(String)
}

fn handle(result: Result) -> String {
    return match result {
        Result::Ok(0)      => "Zero result",
        Result::Ok(value)  => "Got: ${value}",
        Result::Err(msg)   => "Error: ${msg}"
    }
}

print(handle(Result::Ok(0)))            // Zero result
print(handle(Result::Ok(42)))           // Got: 42
print(handle(Result::Err("timeout")))   // Error: timeout
```

A wildcard in a payload position says "there is a value here, and I do not need it." That keeps the arity check—the pattern still only matches a two-payload `Rect`—while binding only what the body uses:

Listing 9.24: Wildcard sub-patterns skip payload positions

```
enum Shape {
    Circle(Float),
    Rect(Float, Float),
    Point
}

fn height(s: Shape) -> Float {
    return match s {
        Shape::Circle(r)  => r * 2.0,
        Shape::Rect(_, h) => h,
        Shape::Point      => 0.0,
        _                 => -1.0
    }
}

print(height(Shape::Circle(4.0)))      // 8
print(height(Shape::Rect(3.0, 7.0)))   // 7
print(height(Shape::Point))            // 0
```

Notice `Shape::Point` in that match: a unit variant is written with no parentheses at all, and it sits happily beside variants that do carry payloads.

9.8.3 Enum Matching with Guards

Guards work seamlessly with enum patterns:

Listing 9.25: Guards on enum patterns

```
enum Temperature {
    Celsius(Float),
    Fahrenheit(Float)
}

fn describe(temp: Temperature) -> String {
    return match temp {
        Temperature::Celsius(c) if c > 100.0    => "Boiling! (${c}C)",
        Temperature::Celsius(c) if c < 0.0      => "Freezing! (${c}C)",
        Temperature::Celsius(c)                  => "Moderate (${c}C)",
        Temperature::Fahrenheit(f) if f > 212.0  => "Boiling! (${f}F)",
        Temperature::Fahrenheit(f)               => "Tolerable (${f}F)"
    }
}

print(describe(Temperature::Celsius(105.0)))    // Boiling! (105C)
print(describe(Temperature::Fahrenheit(72.0)))  // Tolerable (72F)
```

Where Did the `.0` Go?

We wrote `105.0` and Lattice printed `105`. Floats are rendered with C's `%g` format (see `src/value.c`), which drops a trailing `.0` and caps the output at six significant digits—so the circle area we computed a page ago prints as `78.5397` rather than `78.53975`. The stored value is unchanged and still a `Float`; only its printed shape is shorter. Arithmetic always uses the full double-precision value—it is the display that rounds, not the number.

9.9 Exhaustiveness Checking

A match expression is *exhaustive* if it handles every possible value the scrutinee might have. Lattice's compiler includes a static analysis pass (implemented in `src/match_check.c`) that inspects every `match` expression in your program and warns you when coverage is incomplete.

9.9.1 How the Checker Works

The exhaustiveness checker runs after parsing, before code generation. It walks the entire AST—including nested functions, `impl` blocks, and `test` blocks—looking for `match` expressions. For each one, it asks a series of questions:

1. **Are there zero arms?** If so, the checker emits: `warning: match expression on line N has no arms`.
2. **Is there a catch-all arm?** An unguarded wildcard (`_`) or an unguarded binding pattern (`x`) without a phase qualifier covers every remaining case. If one exists, the match is exhaustive and the checker moves on.
3. **Are we matching booleans?** If every pattern is a boolean literal, the checker verifies both `true` and `false` are present.
4. **Everything else**: without a catch-all, the checker recommends: `consider adding a wildcard _ arm`.

Every warning goes to standard error. None of them stop compilation—this is advice, not a gate.

9.9.2 Enum Exhaustiveness

Here is the part that surprises people coming from Rust or Swift. In v0.6.4 the checker does *not* track enum variant coverage. Listing every variant of an enum is good practice, but it will not silence the warning:

Listing 9.26: A fully covered enum still asks for a wildcard

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

let c = Color::Red

let name = match c {
    Color::Red   => "red",
    Color::Green => "green",
    Color::Blue  => "blue"
}

print(name) // red
```

```
warning: non-exhaustive match on line 9: consider adding a wildcard `_` arm
```

Drop the `Color::Blue` arm and you get the same sentence—no mention of which variant went missing. The machinery for per-variant reporting lives in `src/match_check.c`, but its pattern recogniser still looks for the shape enum variants had before they became first-class patterns in v0.3.29, so today's `Color::Red` arms sail past it unrecognised. Every enum match therefore falls through to the generic recommendation.

Do Not Lean on the Checker for Enums

The practical consequence: adding a variant to an enum will *not* produce warnings at the matches that fail to handle it. Nothing tells you at compile time; the match quietly returns `nil` at run time instead. Until per-variant reporting is restored, treat enum coverage as your responsibility—write the wildcard arm, and make it loud:

```
let name = match c {
    Color::Red   => "red",
    Color::Green => "green",
    Color::Blue  => "blue",
    other        => panic("unhandled color: ${other}")
}
```

9.9.3 Boolean Exhaustiveness

When every arm uses a boolean literal, the checker ensures both `true` and `false` are covered:

Listing 9.27: Boolean exhaustiveness

```
let flag = true

// warning: non-exhaustive match: missing `false` case
let label = match flag {
    true => "on"
}

// Fixed:
let label = match flag {
    true  => "on",
    false => "off"
}
```

9.9.4 Limitations of the Checker

The exhaustiveness checker is deliberately pragmatic rather than exhaustive itself. A few limitations to be aware of:

- **No redundancy warnings**: The checker does not report unreachable or redundant arms. If you put a wildcard before specific patterns, no warning is issued.
- **No deep analysis of arrays or structs**: Array and struct patterns are not analyzed for completeness. The checker relies on a wildcard arm for these cases.
- **No enum variant coverage**: as of v0.6.4 the checker does not know which enum variants an arm covers, so it can neither confirm completeness nor name a forgotten variant (Section 9.9.2).
- **Range patterns are not analyzed**: The checker does not verify that a set of range patterns covers the entire integer domain.
- **Guards never count**: A guarded arm can always fail its guard, so it never satisfies the checker—not even a guarded wildcard.

A Good Habit

Even when the checker does not require it, adding a wildcard arm with a meaningful response (or an error) is good defensive programming. Future changes to an enum or data format are less likely to introduce silent bugs if there is a catch-all that handles the unexpected.

9.10 Phase-Qualified Patterns

Lattice's phase system (explored in depth in Chapter 11) assigns every value a *phase*—fluid, crystal, or unphased. Pattern matching can discriminate on phase by prefixing a pattern with a phase qualifier:

Listing 9.28: Phase-qualified patterns

```
flux temperature = 72
fix boiling_point = 212

let status = match temperature {
    crystal _ => "This value is crystallized",
    fluid t   => "Fluid value: ${t}"
}

print(status) // Fluid value: 72

let bp_status = match boiling_point {
    crystal t => "Crystal: ${t}",
    fluid _   => "Fluid"
}

print(bp_status) // Crystal: 212
```

The `fluid` qualifier matches values in the fluid or unphased state (which is the default). The `crystal` qualifier matches only values that have been crystallized with `fix` or `freeze`.

Phase Qualifiers and Exhaustiveness

A phase-qualified wildcard or binding is *not* considered a catch-all by the exhaustiveness checker. The pattern `crystal _` only matches crystal values, so it cannot guarantee coverage of all inputs. You still need an unqualified catch-all arm.

Phase-qualified patterns are a niche feature. They become useful in generic code that must behave differently depending on whether a value is mutable or immutable, which we will explore further in Chapter 11.

9.11 Putting It All Together

Let us build a more realistic example that combines multiple pattern types. Imagine we are writing a command-line argument parser:

Listing 9.29: A command-line argument processor

```
fn process_args(args: Array) -> String {
    return match args {
        []                    => "No arguments provided. Use --help.",
        ["--help"]            => "Usage: app [options] [files...]",
        ["--version"]         => "v2.1.0",
        ["--output", path]    => "Output will be written to ${path}",
        ["--verbose", ...rest] => {
            let files = rest.join(", ")
            "Verbose mode, processing: ${files}"
        },
        [cmd, ...rest] => "Unknown command: ${cmd}"
    }
}

print(process_args([]))
// No arguments provided. Use --help.

print(process_args(["--help"]))
// Usage: app [options] [files...]

print(process_args(["--output", "/tmp/out.txt"]))
// Output will be written to /tmp/out.txt

print(process_args(["--verbose", "a.txt", "b.txt"]))
// Verbose mode, processing: a.txt, b.txt
```

Here is another example that processes a stream of events using enum variants:

Listing 9.30: Processing events with enum matching

```
enum Event {
    Click(Int, Int),
    KeyPress(String),
    Resize(Int, Int),
    Close
}

fn handle_event(event: Event) -> String {
    return match event {
        Event::Click(x, y) if x < 0 => "Click out of bounds",
        Event::Click(0, 0)          => "Click at origin",
        Event::Click(x, y)          => "Click at (${x}, ${y})",
        Event::KeyPress("q")        => "Quit requested",
        Event::KeyPress("h")        => "Help requested",
        Event::KeyPress(key)        => "Key: ${key}",
        Event::Resize(w, h)         => "Window is now ${w}x${h}",
        Event::Close                => "Goodbye!"
    }
}

print(handle_event(Event::Click(100, 200)))
// Click at (100, 200)

print(handle_event(Event::KeyPress("q")))
// Quit requested

print(handle_event(Event::Close))
// Goodbye!
```

And finally, an example that matches struct data with guards:

Listing 9.31: Struct matching for access control

```
struct Request {
    method: String,
    path: String,
    authenticated: Bool
}

fn route(req: Request) -> String {
    return match req {
        {method: "GET", path: "/"}          => "Home page",
        {method: "GET", path: "/health"}    => "OK",
        {method: "GET", path}               => "Serving ${path}",
        {method: "POST", authenticated: false} => "401 Unauthorized",
        {method: "POST", path}              => "Processing POST to ${path}",
        {method}                            => "405 Method Not Allowed: ${method}"
    }
}

let req = Request { method: "POST", path: "/api/data", authenticated: true }
print(route(req)) // Processing POST to /api/data

let bad_req = Request { method: "POST", path: "/api/data", authenticated: false }
print(route(bad_req)) // 401 Unauthorized
```

9.12 How Match Compiles

Understanding how `match` compiles helps you reason about performance and appreciate the shape of the design. There is no single "match" instruction; the compiler in `src/stackcompiler.c` lowers a match expression into ordinary comparisons and jumps, helped along by a handful of small pattern-testing opcodes.

9.12.1 The Hidden Scrutinee Slot

The compiler's central move is to evaluate the scrutinee *exactly once* and park it in an anonymous local slot—you can see `add_local("")` do this at the top of `compile_match_expression`. Every arm then begins with an `OP_GET_LOCAL` that reloads a fresh copy from that slot.

This matters for more than tidiness. Because the scrutinee lives in a named slot rather than floating on the stack, a side-effecting scrutinee like `match next_token() { ... }` fires its side effect once no matter how many arms are tested, and the arms cannot disturb each other's view of the value.

9.12.2 Compilation by Pattern Type

Literal patterns push the literal constant and compare with `OP_EQ`.

Wildcard patterns without a phase qualifier push `OP_TRUE` directly—they always match.

Range patterns push the two endpoints and emit `OP_MATCH_RANGE`, a dedicated opcode that performs the inclusive comparison while leaving the candidate value in place beneath the result.

Phase qualifiers become `OP_IS_CRYSTAL` or `OP_MATCH_FLUID` (the latter answering true for both fluid and unphased values, which is why `fluid` is the broader of the two qualifiers).

Binding patterns reload the scrutinee and create a local variable slot. Guard compilation emits the guard expression, and the result is tested with `OP_JUMP_IF_FALSE`.

Complex patterns (arrays, structs, enums) use a two-phase approach:

1. **Structural check**: `OP_MATCH_TYPE` verifies the runtime tag (array, struct, enum) without disturbing the candidate; then length, `OP_HAS_FIELD` field-presence tests, and literal sub-patterns are checked. Every step is chained with a short-circuit `OP_JUMP_IF_FALSE`, so the first failure skips all remaining work.

2. **Binding extraction**: only once the structural check passes. Array elements come out with `OP_INDEX`, struct fields with `OP_GET_FIELD`, and enum payloads by invoking the `.payload()` method and indexing the resulting array. Sub-patterns recurse through the same two phases.

After the matched arm's body executes, an `OP_JUMP` skips over all remaining arms. At the very end, a fallback emits `OP_POP` (to release the scrutinee slot) and `OP_NIL`—the no-match value we met back in Section 9.1.2.

The register VM lowers the same patterns with its own trio of helpers, `ROP_MATCH_TYPE`, `ROP_MATCH_FIELD`, and `ROP_MATCH_PHASE`, but the staging is identical: structural test first, extraction second.

Performance Implications

Because each arm is compiled as a linear sequence of checks and jumps, match expressions are evaluated in $O(n)$ time where n is the number of arms. The compiler does not build a jump table or decision tree. For small match expressions (the common case) that is perfectly efficient, and for matches with many literal arms the short-circuit jumps keep things quick by skipping unnecessary work.

Match Survives a Round Trip Through Bytecode

Since bytecode format v3, which first shipped in v0.6.4, `match` lowers entirely to opcodes with no leftover dependency on the syntax tree—as do `select` and `defer`. That means a program full of pattern matching compiles and re-runs cleanly:

```
clat compile events.lat -o events.latc
clat events.latc
```

Earlier releases could not serialize these forms at all. Chapter 34 covers the format in detail.

9.12.3 Match in Expression Position

There is one shape the compiler will refuse. When a `match` appears as an *operand*—an argument, an array element, one side of a +—the default VM compiles it inside a synthetic zero-argument closure that is invoked on the spot. The fresh call frame keeps the hidden scrutinee slot from colliding with operand temporaries already sitting on the stack.

That synthetic frame is also a wall. A `return`, or a `break`/`continue` bound to a loop outside the match, cannot cross it, so the compiler rejects the program rather than mis-compile it:

Listing 9.32: return cannot escape a match used as an operand

```
fn f(n: Int) -> String {
    let label = "n=" + match n { 1 => { return "early" }, _ => "other" }
    return label
}

print(f(1))
// compile error: 2:1: return/break/continue inside a match used as an
// operand is not supported (bind the match result with 'let' first)
```

The fix is the one the message names—give the match its own `let`, where it is no longer an operand:

Listing 9.33: Bind the match first

```
fn f(n: Int) -> String {
    let tail = match n { 1 => "one", _ => "other" }
    return "n=" + tail
}

print(f(1)) // n=one
```

Backend Divergence

This guard is specific to the default stack VM. Under `--tree-walk` and `--regvm` the first listing above compiles and prints `early`. Treat the stack VM's rule as the language rule: a program that relies on jumping out of an operand-position match will stop working the moment it runs on the default backend.

9.13 Common Patterns and Idioms

9.13.1 Replacing Long if/else Chains

Any time you find yourself writing three or more `if`/`else` branches that all test the same variable, consider using `match` instead:

Listing 9.34: Before and after: if/else to match

```
// Before
fn day_type(day: String) -> String {
    if day == "Saturday" {
        return "weekend"
    } else if day == "Sunday" {
        return "weekend"
    } else {
        return "weekday"
    }
}

// After
fn day_type(day: String) -> String {
    return match day {
        "Saturday" => "weekend",
        "Sunday"   => "weekend",
        _          => "weekday"
    }
}
```

The `match` version is more compact, makes the structure of the decision visible at a glance, and ensures every case is handled.

9.13.2 Extracting from Nested Structures

Match excels at pulling values out of nested data:

Listing 9.35: Extracting nested data

```
enum Response {
    Success(Array),
    Error(String)
}

fn first_item(resp: Response) -> String {
    return match resp {
        Response::Success([first, ...rest]) => "First: ${first}",
        Response::Success([])               => "Empty results",
        Response::Error(msg)                => "Failed: ${msg}",
        _                                   => "Unknown"
    }
}

print(first_item(Response::Success([10, 20, 30])))
// First: 10

print(first_item(Response::Success([])))
// Empty results

print(first_item(Response::Error("timeout")))
// Failed: timeout
```

9.13.3 State Machines

Enums and match are a natural way to model state machines. Each state is a variant, and each transition is a match arm:

Listing 9.36: A state machine with match

```
enum ConnectionState {
    Disconnected,
    Connecting(String),
    Connected(String),
    Error(String)
}

fn next_state(state: ConnectionState, action: String) -> ConnectionState {
    return match state {
        ConnectionState::Disconnected if action == "connect" => {
            ConnectionState::Connecting("server.example.com")
        },
        ConnectionState::Connecting(host) if action == "success" => {
            ConnectionState::Connected(host)
        },
        ConnectionState::Connecting(_) if action == "fail" => {
            ConnectionState::Error("Connection refused")
        },
        ConnectionState::Connected(_) if action == "disconnect" => {
            ConnectionState::Disconnected
        },
        other => other
    }
}
```

9.13.4 The Option Pattern

A very common idiom in Lattice is using an `Option`-like enum to represent values that may or may not exist, and matching to handle both cases:

Listing 9.37: The Option pattern

```
enum Option {
    Some(any),
    None
}

fn find_user(id: Int) -> Option {
    if id == 1 {
        return Option::Some("Alice")
    }
    return Option::None
}

let user = find_user(1)

let greeting = match user {
    Option::Some(name) => "Hello, ${name}!",
    Option::None       => "User not found"
}

print(greeting) // Hello, Alice!
```

This pattern appears throughout Lattice code and is explored further in Chapter 10, where we combine it with `Result` types for robust error handling.

9.14 Exercises

1. **FizzBuzz with match.** Write a function `fn fizzbuzz(n: Int) -> String` that uses a `match` expression with guards to return `"FizzBuzz"` when `n` is divisible by both 3 and 5, `"Fizz"` when divisible by 3, `"Buzz"` when divisible by 5, and the number as a string otherwise. Hint: use a binding pattern with guards.

2. **List operations.** Write a function `fn sum_list(items: Array) -> Int` that computes the sum of an integer array using `match` with array destructuring and rest patterns. If the array is empty, return 0. If it has one element, return that element. If it has more, extract the first element and recursively sum the rest.

3. **Shape perimeter.** Define an enum `Shape` with variants `Circle(Float)`, `Rectangle(Float, Float)`, and `Square(Float)`. Write a function `fn perimeter(s: Shape) -> Float` that uses pattern matching to compute the perimeter. What happens if you forget to handle the `Square` variant?

4. **HTTP router.** Define a struct `Request` with fields `method: String` and `path: String`. Write a function `fn route(req: Request) -> String` that uses struct pattern matching to return different responses for `GET /`, `GET /about`, `POST /login`, and a catch-all `404` response.

5. **Expression evaluator.** Define an enum for arithmetic expressions:

```
enum Expr {
    Num(Int),
    Add(any, any),
    Mul(any, any)
}
```

   Write a function `fn evaluate(e: Expr) -> Int` that recursively evaluates an expression tree using pattern matching. For example, `Expr::Add(Expr::Num(2), Expr::Num(3))` should evaluate to 5.

What's Next Pattern matching gives us the tools to inspect data and branch on its structure. But what happens when things go wrong—when a file is missing, a network request times out, or a function receives unexpected input? In Chapter 10, we will explore Lattice's approach to error handling: the `error()` and `is_error()` functions, `try`/`catch` blocks, the `?` propagation operator, `defer` for cleanup, and how to design programs that fail gracefully.

Chapter 10

Error Handling

Things go wrong. Files disappear between the time you check for them and the time you open them. Network requests time out. Users type letters where numbers belong. A language that pretends errors do not happen is a language that produces programs crashing at 3 a.m. with no explanation.

Lattice takes errors seriously without making them painful. It gives you multiple tools—each suited to a different situation—and lets you choose the right one for the job. At the most basic level, the `error()` function creates error values that you can test with `is_error()`. When you need structured recovery, `try`/`catch` intercepts errors and hands you a detailed error map with a message, a line number, and a stack trace. For code that uses `Result` maps, the `?` operator unwraps successes and propagates failures in a single character. And `defer` ensures that cleanup code runs no matter how a scope exits—normally or through an error.

Listing 10.1: A quick tour of error handling

```
import "lib/fn" as F

fn safe_divide(a: Float, b: Float) -> Map {
    if b == 0.0 {
        return F.err("division by zero")
    }
    return F.ok(a / b)
}

let result = safe_divide(10.0, 0.0)

let message = match result.get("tag") {
    "ok" => "Result: ${result.get("value")}",
    _    => "Error: ${result.get("value")}"
}

print(message) // Error: division by zero
```

This chapter builds from the ground up. We start with the simplest error tools, progress through `try`/`catch` and the `?` operator, explore `defer` for resource cleanup, revisit contracts from Chapter 6, and finish with patterns for designing programs that fail gracefully.

Struct Patterns Do Not Match Maps

It is tempting to write `match result { {tag: "ok", value} => ... }` and destructure the Result directly. It will not fire. A struct pattern compiles to a runtime type test for `Struct` (`OP_MATCH_TYPE` with the struct tag), and a Result is a `Map`—so every such arm is skipped and the match falls through to the wildcard, or to `nil`.
Match on `result.get("tag")` as we did above, or use the `is_ok()`/`is_err()` helpers from Section 10.2.1. Struct patterns are for real structs, which we cover in Chapter 15.

10.1 Error Values: `error()` and `is_error()`

The most direct way to signal a problem in Lattice is to return an error value. The built-in `error()` function takes a message and produces a value that `is_error()` recognizes:

Listing 10.2: Creating and detecting error values

```
let bad = error("something went wrong")

print(is_error(bad))   // true
print(is_error("ok"))  // false
print(is_error(42))    // false
```

What is that value, exactly? On both bytecode backends, `error("msg")` builds a two-key `Map`—you can read the whole implementation in `native_error` in `src/runtime.c`, and it is four lines long:

Listing 10.3: An error value is a Map

```
let bad = error("disk full")

print(typeof(bad))       // Map
print(bad)               // {"tag": err, "value": disk full}
print(bad.get("tag"))    // err
print(bad.get("value"))  // disk full
```

That is the same shape as the `Result` convention we are about to meet in Section 10.2: on the bytecode backends `error(m)` and `err(m)` produce values that compare equal, and `is_error()` asks whether it is looking at a Map whose `"tag"` is `"err"`. There, the two vocabularies are one vocabulary.

The Tree-Walker's Older Representation

Under `--tree-walk`, `error()` still returns the pre-v0.5.0 representation: a `String` carrying the internal marker `EVAL_ERROR:` (so `print(bad)` shows `EVAL_ERROR:disk full` and `.get()` is not available on it).

What survives that difference is the narrow question: `is_error(error(m))` answers `true` on all three backends. What does not survive is mixing the vocabularies. Hand an `err()` Map to `is_error()` and the tree-walker answers `false` where both VMs answer `true`, because there the two constructors build different types (Section 10.2.1). So: ask `is_error()` about values `error()` produced, do not inspect the representation, and do not assume the two vocabularies meet.

Listing 10.4: Using error() in a function

```
fn to_number(s: String) -> any {
    let digits = "0123456789"
    for ch in s.chars() {
        if !digits.contains(ch) {
            return error("invalid character: ${ch}")
        }
    }
    return parse_int(s)
}

let result = to_number("12x4")

if is_error(result) {
    print("Parse failed")
} else {
    print("Got: ${result}")
}
// Parse failed
```

This approach works, but it has a limitation: the caller must remember to check `is_error()` on every return value. Forget once, and a failure sails on as if it were data. For more robust error handling, Lattice offers two better alternatives: the `Result` type (Section 10.2) and `try`/`catch` (Section 10.3).

error() Does Not Throw

Returning `error("...")` produces a *value*. It does not unwind, it does not fire an exception handler, and wrapping the call in `try`/`catch` will not catch anything—the `try` block evaluates to the error value. When you want the stack to unwind, use `assert()`, `panic()`, or a `require` contract (Section 10.7).

10.2 The Result Type

Lattice's standard library (in `lib/fn.lat`) provides a convention for representing outcomes that might succeed or fail: the `Result` type. A Result is a plain `Map` with two keys:

- `"tag"`: either `"ok"` or `"err"`
- `"value"`: the success value or the error message

These helpers do not live in the global namespace—they come from `lib/fn.lat`, so every listing in this section starts with an import:

Listing 10.5: Creating Result values

```
import "lib/fn" as F

let success = F.ok(42)
let failure = F.err("file not found")

print(success) // {"tag": ok, "value": 42}
print(failure) // {"tag": err, "value": file not found}
```

Imports Resolve From the Working Directory

`import "lib/fn"` is resolved relative to the directory you run `clat` from, not relative to the script. Run these examples from the root of a Lattice checkout, or give the import an absolute path.

Result Convention

A *Result* is a Map with a `"tag"` field set to `"ok"` or `"err"` and a `"value"` field carrying the payload. The standard library functions `ok(value)` and `err(message)` construct Result maps. The `?` operator and functions like `is_ok()`, `is_err()`, `unwrap()`, and `unwrap_or()` work with this convention.

10.2.1 Working with Results

The standard library provides several functions for inspecting and transforming Results:

Listing 10.6: Result helper functions

```
import "lib/fn" as F

let result = F.ok(100)

print(F.is_ok(result))   // true
print(F.is_err(result))  // false

// Extract the value (throws if err!)
print(F.unwrap(result))  // 100

// Extract with a default fallback
let r2 = F.err("timeout")
print(F.unwrap_or(r2, 0)) // 0
```

On the bytecode backends, `error()` builds exactly the same Map that `err()` does, so the two vocabularies interoperate without conversion:

Listing 10.7: error() and err() are the same value on the bytecode backends

```
import "lib/fn" as F

print(error("boom") == F.err("boom"))  // true (default VM and --regvm)
print(is_error(F.err("boom")))         // true (default VM and --regvm)
print(F.is_err(error("boom")))         // true (default VM and --regvm)
```

Mixing error() and err() Breaks the Tree-Walker

Run that listing with `--tree-walk` and not one line agrees. There `error()` is still a marked `String` rather than a Map (Section 10.1), so the first two lines print `false`, and the third does not print at all—`F.is_err()` reaches for `.get()` on a String and stops the program with `.get() is not defined on non-struct`.

Pick one vocabulary per program and stay inside it: `error()` with `is_error()`, or `err()` with `is_err()`. Code that crosses between them runs on the two VMs and breaks on the tree-walker.

unwrap() on Errors

Calling `unwrap()` on an `err` Result raises a catchable assertion failure whose message is `assertion failed: unwrap() called on Err: timeout`. Use `unwrap()` only when you are certain the Result is `ok`, or prefer `unwrap_or()` for a safe fallback.

You can transform Results without unwrapping them using `map_result()` and `flat_map_result()`:

Listing 10.8: Transforming Results

```
import "lib/fn" as F

let result = F.ok(5)

// Apply a function to the ok value
let doubled = F.map_result(result, |x| x * 2)
print(F.unwrap(doubled)) // 10

// Chain operations that return Results
let chained = F.flat_map_result(F.ok(10), |x| {
    if x > 100 {
        return F.err("too large")
    }
    return F.ok(x * x)
})
print(F.unwrap(chained)) // 100
```

`map_result()` applies a function to the value inside an `ok` and rewraps the result; if the input is `err`, it passes through unchanged. `flat_map_result()` does the same but expects the function to return a Result itself, avoiding double-wrapping.

10.3 try/catch

When an operation might throw a runtime error—a type mismatch, a division by zero, an assertion failure, or a contract violation—you can intercept it with `try/catch`:

Listing 10.9: Basic try/catch

```
let result = try {
    let x = 10 / 0
    x
} catch e {
    print("Caught: ${e.get("message")}")
    let fallback = 0 - 1
    fallback
}

print(result)
// Caught: division by zero
// -1
```

Do Not End a Block With a Bare Negative Literal

That `let fallback = 0 - 1` looks like ceremony, and it is there for a reason. Lattice does not treat a newline as a statement terminator, so writing

```
print("Caught: ${e.get("message")}")
-1
```

parses as `print(...) - 1`—subtracting one from `print`'s Unit return, which fails with `operands must be numbers for '-'`. A leading `-`, `+`, `(`, or `[` on a new line continues the previous expression. Bind the value to a name first, or parenthesize it.

Notice that the message we caught is the bare text `division by zero`. Had we let that error escape to the top level, it would have been printed with a prefix that tells you which backend produced it:

Backend	Uncaught error prefix
default (stack VM)	`vm error: division by zero`
`--tree-walk`	`error: division by zero`
`--regvm`	`regvm error: division by zero`

Reading Error Prefixes

The prefix is the reporter, not the error. The same failure carries the same core message on all three backends—only the tag in front changes, and it tells you which execution engine was running. Errors raised before execution begins are tagged by stage instead: `error:` for lexer and parser problems, `compile error:` (or `regvm compile error:`) for things the compiler rejects. Throughout this book we quote the core message. When you see `division by zero` in a listing and `vm error: division by zero` in your terminal, nothing has gone wrong.

10.3.1 try/catch Is an Expression

Like `if`/`else` and `match`, `try`/`catch` is an expression. The value of the `try` block (if it succeeds) or the `catch` block (if an error occurs) becomes the value of the entire expression:

Listing 10.10: try/catch produces a value

```
fn safe_get(items: Array, index: Int) -> any {
    return try {
        items[index]
    } catch e {
        nil
    }
}

print(safe_get([10, 20, 30], 1))  // 20
print(safe_get([10, 20, 30], 99)) // nil
```

10.3.2 The Error Map

The variable after `catch` (always required—you cannot omit it) is bound to a structured error `Map` with three fields:

Key	Type	Description
`"message"`	String	The error message
`"line"`	Int	Source line where the error occurred
`"stack"`	Array	Stack trace (array of strings)

Listing 10.11: Inspecting the error map

```
try {
    assert(false, "something broke")
} catch e {
    print(e.get("message"))
    print(e.get("line"))
    print(e.get("stack"))
}
// assertion failed: something broke
// 2
// [<script> at line 2]
```

Note the message: `assert()` prefixes your text with `assertion failed:`. If you are comparing an error message against a literal in a test, compare against the whole thing.

The stack trace is an array of strings, each in the format `"function_name() at line N"`, innermost frame first. For the top-level script, it appears as `"<script> at line N"`. For closures without names, it appears as `"<closure> at line N"`.

Listing 10.12: Reading the stack trace

```
fn inner() -> any {
    assert(false, "deep error")
    return 0
}

fn middle() -> any {
    return inner()
}

try {
    middle()
} catch e {
    print("msg:  ${e.get("message")}")
    print("line: ${e.get("line")}")
    for entry in e.get("stack") {
        print("  ${entry}")
    }
}
// msg:  assertion failed: deep error
// line: 2
//   inner() at line 2
//   middle() at line 7
//   <script> at line 11
```

The trace reads bottom-up in the usual sense: `inner()` raised it, `middle()` called `inner()`, and the script called `middle()` from inside the `try`.

Traces Are Richest on the Bytecode VMs

The default VM and `--regvm` record real function names and line numbers—including for code loaded from a compiled `.latc` file, which used to show `<closure>` everywhere. The tree-walker keeps a thinner record: the same run reports `line` as `0` and a stack of `in inner()`, `in middle()` with no line numbers and no `<script>` frame.

Uncaught errors print the same information to standard error, formatted for reading. Here is a frozen array being written to from inside a function:

Listing 10.13: An uncaught error and its printed trace

```
flux a = [1, 2]
freeze(a)

fn f() -> any {
    a[0] = 9
    return 0
}

f()
```

```
vm error: cannot modify a frozen value
stack trace (most recent call last):
  [line 9] in <script>
  [line 5] in f()
```

Read it as the header says—most recent call last—so the bottom line, `f()`, is where the error fired.

10.3.3 What Can Be Caught?

The `try`/`catch` mechanism catches a wide range of runtime errors:

- Division by zero
- Index out of bounds
- Undefined variables and calls to methods a type does not have
- Type mismatches (parameter and return type checks)
- Wrong argument counts (`expected 2 arguments but got 1`)
- Assertion failures (`assert()`, `assert_eq()`, etc.)
- Contract violations (`require` and `ensure`)
- Phase violations—writing to a frozen or sublimated value
- Freeze and anneal contract failures
- Errors raised inside a closure passed to `Array.map`—but not inside the other combinators, which unwind only under `--tree-walk` (Section 7.2.7)

- Runaway recursion, which the bytecode VMs report as `stack overflow (too many nested calls)` rather than crashing
- Parse and format failures from `json_parse`, `base64_decode`, `to_int`, and friends
- Missing module exports

That `map` entry is worth a demonstration, because it used to be undefined behavior. An error thrown from inside a `map` callback now propagates cleanly to the enclosing handler on every backend:

Listing 10.14: Errors escape from a map callback

```
let outcome = try {
    [1, 0, 2].map(|x| { 12 / x })
} catch e {
    e.get("message")
}

print(outcome) // division by zero
```

Read that entry narrowly. `map` is the only array combinator that reaches your handler on all three backends; `filter`, `reduce`, `any`, `all`, `find`, and the rest end the process on the two bytecode VMs instead of unwinding, and only `--tree-walk` recovers from them. Section 7.2.7 carries the full list, and Section 26.5 does the same for lazy `iter(...)` chains.

What `try`/`catch` will *not* catch is anything that fails before your program runs—lexer errors, parse errors, and compile errors. Those are tagged by stage rather than by backend: `error:` for lexer and parser problems on all three engines, and `compile error:` (or `regvm compile error:`) for what the compilers reject. They go to standard error and the process exits. There is no handler installed yet, so there is nothing for `catch` to intercept.

Under the hood, `try` compiles to `OP_PUSH_EXCEPTION_HANDLER`, which registers a handler with the VM. The handler records the catch block's instruction pointer, the current call frame index, and a snapshot of the stack top. When an error fires (via the `VM_ERROR` macro or `OP_THROW`), the VM looks for the nearest handler, unwinds the call stack to the handler's frame, restores the stack, and jumps to the catch block with the error map pushed on top. You can see this mechanism in `src/stackvm.c`.

Handler Limit

The VM supports up to 64 nested exception handlers (`STACKVM_HANDLER_MAX`). This is more than enough for any reasonable program. If you exceed this limit, the VM throws its own error: `"too many nested exception handlers"`.

10.4 The ? Operator

The ? operator is Lattice's most concise error-handling tool. It works with Result maps (Section 10.2): if the Result is `ok`, the ? extracts the inner value; if it is `err`, it immediately returns the error from the current function.

Listing 10.15: The ? operator in action

```
import "lib/fn" as F

fn halve(n: Int) -> Map {
    if n % 2 != 0 {
        return F.err("${n} is not even")
    }
    return F.ok(n / 2)
}

fn quarter(n: Int) -> Map {
    let half = halve(n)?    // propagates err
    let q = halve(half)?    // propagates err
    return F.ok(q)
}

print(quarter(20)) // {"tag": ok, "value": 5}
print(quarter(21)) // {"tag": err, "value": 21 is not even}
print(quarter(10)) // {"tag": err, "value": 5 is not even}
```

That third call is the interesting one: `halve(10)` succeeds and hands back `5`, then the second ? catches the failure and returns it from `quarter` untouched. The error that surfaces names the value that actually failed.

Without ?, you would need to check each Result by hand:

Listing 10.16: Manual error checking (the verbose way)

```
import "lib/fn" as F

fn quarter(n: Int) -> Map {
    let half_result = halve(n)
    if F.is_err(half_result) {
        return half_result
    }
    let half = F.unwrap(half_result)

    let q_result = halve(half)
    if F.is_err(q_result) {
        return q_result
    }
    return F.ok(F.unwrap(q_result))
}
```

The ? operator compresses each four-line check-and-unwrap into a single postfix character.

10.4.1 How ? Works

At compile time, `expr?` compiles to the expression followed by a single `OP_TRY_UNWRAP` opcode. At runtime, this opcode inspects the top of the stack:

1. If the value is a Map with `"tag"` = `"ok"`: extract the `"value"` field and replace the stack top with it. Execution continues normally.

2. If the value is a Map with `"tag"` = `"err"`: immediately return from the current function with the error map as the return value. No exception handler is needed; the operator uses the normal return path.

3. If the value is not a conforming Result map: throw a runtime error: `"'?' operator requires a result map with \{tag: \"ok\"`ërr; value: ...}"|.

? Requires Result Maps

The ? operator only works with maps that follow the Result convention. Using ? on a plain integer, string, or non-conforming map throws a runtime error—and this is one of the few messages whose wording differs across backends, so match on behavior rather than on text:

```
let r = try { let x = 5? ; x } catch e { e.get("message") }
print(r)

// default:      '?' operator requires a result map with
//               {tag: "ok"|"err", value: ...}
// --tree-walk:  ? operator requires a Result map (got non-Map value)
// --regvm:      '?' operator requires a Result value, got Int
```

Because `error()` now builds a conforming err Map (Section 10.1), ? works on `error()` values too on the bytecode backends. Under `--tree-walk`, where `error()` is still a marked String, it does not—so reach for `err()` when you intend to propagate.

10.4.2 Chaining ?

Because ? unwraps the ok value in place, you can stack several of them into a single linear pipeline:

Listing 10.17: Chaining the ? operator

```
import "lib/fn" as F

fn validate(input: Map) -> Map {
    if input.get("name") == nil {
        return F.err("missing name")
    }
    return F.ok(input)
}

fn transform(input: Map) -> Map {
    flux out = input
    out["name"] = input.get("name").to_upper()
    return F.ok(out)
}

fn save(record: Map) -> Map {
    return F.ok(record.get("name"))
}

fn process_data(input: Map) -> Map {
    let validated = validate(input)?
    let transformed = transform(validated)?
    let saved = save(transformed)?
    return F.ok("Processed ${saved}")
}

flux good = Map::new()
good["name"] = "rosalind"
print(process_data(good))       // {"tag": ok, "value": Processed ROSALIND}
print(process_data(Map::new())) // {"tag": err, "value": missing name}
```

Each ? either continues with the unwrapped value or short-circuits the entire function, returning the first error encountered. Errors are handled without a single `if`.

10.5 panic()

When something goes so catastrophically wrong that recovery makes no sense, `panic()` signals a fatal error:

Listing 10.18: Using panic() for unrecoverable errors

```
fn initialize(config: Map) -> any {
    let db_url = config.get("database_url")
    if db_url == nil {
        panic("Cannot start without a database URL")
    }
    return db_url
}

flux good = Map::new()
good["database_url"] = "postgres://localhost/app"
print(initialize(good))  // postgres://localhost/app

initialize(Map::new())
// vm error: Cannot start without a database URL
// stack trace (most recent call last):
//   [line 12] in <script>
//   [line 4] in initialize()
```

`panic()` is semantically different from `error()`: it communicates that the program has reached a state that should never happen—an invariant violation, a logic error, or a missing critical resource. Use `error()` and Result types for expected failures (network timeouts, invalid user input); reserve `panic()` for bugs and impossible situations.

Mechanically, though, `panic()` raises an ordinary runtime error, which means a `try`/`catch` *will* intercept it, message and all. That is a property of the implementation, not an invitation. The distinction between the two lives in your head and in the heads of the people reading your code: one says "this can happen, handle it," the other says "if this happens the program is already wrong."

When to panic()

Good uses of `panic()`: unreachable code branches, failed internal invariants, missing required configuration at startup. Bad uses: user input validation, file-not-found errors, network failures—these are expected and should be handled with Results or `try`/`catch`.

10.6 defer

Resources need cleanup. Files should be closed, locks released, temporary directories removed. The problem with manual cleanup is that errors can cause early exits, skipping the cleanup code. `defer`

solves this by guaranteeing that a block of code runs when the enclosing scope exits, no matter how it exits:

Listing 10.19: defer ensures cleanup runs

```
fn line_count(path: String) -> Int {
    let text = read_file(path)
    defer { print("finished reading ${path}") }

    // Even if this throws, the defer block runs
    return text.split("\n").len()
}

write_file("/tmp/notes.txt", "alpha\nbeta\ngamma")
print(line_count("/tmp/notes.txt"))
// finished reading /tmp/notes.txt
// 3
```

10.6.1 Execution Order: LIFO

When multiple `defer` blocks are registered in the same scope, they execute in *last-in, first-out* (LIFO) order—the most recently deferred block runs first:

Listing 10.20: Defers run in reverse order

```
fn demo() -> any {
    defer { print("First deferred, runs last") }
    defer { print("Second deferred, runs first") }
    print("Function body")
    return nil
}

demo()
// Function body
// Second deferred, runs first
// First deferred, runs last
```

This LIFO ordering is intentional: resources acquired later typically depend on resources acquired earlier, so they should be released first.

10.6.2 Scope-Aware Execution

Defers are tied to their enclosing scope. When a scope exits—whether it is a function body, a block, or a loop iteration—only the defers registered in that scope run:

Listing 10.21: Defer and scope boundaries

```
fn outer() -> any {
    defer { print("outer defer") }

    for i in 0..3 {
        defer { print("loop defer ${i}") }
        print("iteration ${i}")
    }

    print("after loop")
    return nil
}

outer()
// iteration 0
// loop defer 0
// iteration 1
// loop defer 1
// iteration 2
// loop defer 2
// after loop
// outer defer
```

Each loop iteration creates and destroys a scope, so the loop's defer runs at the end of each iteration, not at the end of the function.

10.6.3 Defer and Errors

Defers run even when an error causes an early exit. This is the whole point—cleanup that only runs on the happy path is not reliable cleanup:

Listing 10.22: Defer runs on error paths

```
fn risky() -> any {
    defer { print("cleanup always runs") }

    print("about to fail")
    assert(false, "intentional failure")
    print("this never prints")
    return nil
}

try {
    risky()
} catch e {
    print("caught: ${e.get("message")}")
}
// about to fail
// cleanup always runs
// caught: intentional failure
```

10.6.4 Defer and Return Values

An important detail: deferred blocks do not affect the return value of the enclosing function. The return value is determined *before* defers execute, and it is preserved across all defer block executions:

Listing 10.23: Defer does not alter return values

```
fn compute() -> Int {
    defer { print("cleaning up") }
    return 42
}

let result = compute()
// cleaning up
print(result) // 42
```

Under the hood, when `OP_DEFER_RUN` executes, the VM saves the current top-of-stack value (the return value), runs each defer body, then restores the saved value. Defer bodies share the parent frame's local variables, so they can read (and even modify) locals, but they cannot change what the function returns.

10.6.5 How Defer Compiles

The defer body is compiled inline in the function's bytecode. An `OP_DEFER_PUSH` instruction records the body's location and scope depth, then a jump skips past the body during normal execution. When `OP_DEFER_RUN` fires (at scope exit or function return), the VM creates a lightweight wrapper and executes the body as a sub-frame.

At every return point in a compiled function, the compiler emits the sequence: (1) return type check, (2) ensure postcondition checks, (3) `OP_DEFER_RUN` with scope depth 0 (run all defers), (4) `OP_RETURN`. This guarantees defers run regardless of which return path the function takes.

10.7 Contracts Revisited

In Chapter 6 we introduced `require` and `ensure` contracts. Now that we understand error handling, let us see how they fit into the bigger picture.

10.7.1 require: Preconditions

A `require` contract checks a condition at function entry. If the condition is false, it throws an error that can be caught by `try`/`catch`:

Listing 10.24: require throws catchable errors

```
fn withdraw(balance: Float, amount: Float) -> Float
    require amount > 0.0, "amount must be positive"
    require amount <= balance, "insufficient funds"
{
    return balance - amount
}

// Successful call
print(withdraw(100.0, 30.0)) // 70.0

// Failed precondition
try {
    withdraw(100.0, -5.0)
} catch e {
    print(e.get("message"))
    // require failed in 'withdraw': amount must be positive
}
```

The error message always includes the function name and the custom message (or `"condition not met"` if no message was provided). Under the hood, the compiler emits the condition expression followed by `OP_JUMP_IF_TRUE` (skip if ok) or `OP_THROW` with the formatted message string.

10.7.2 ensure: Postconditions

An `ensure` contract checks the return value of a function. It takes a closure that receives the return value and must return a truthy value:

Listing 10.25: ensure validates return values

```
fn half(n: Int) -> Int
    ensure |result| { result * 2 == n }, "result must be half of input"
{
    return n / 2
}

print(half(10)) // 5

// With an odd number, integer division truncates
try {
    half(7)
} catch e {
    print(e.get("message"))
    // ensure failed in 'half': result must be half of input
}
```

The ensure check runs at every return point—both explicit `return` statements and the implicit trailing expression. The compiler inserts the check *before* defers run, so the order at each return point is: type check → ensure checks → defer execution → actual return.

10.7.3 Combining Contracts with try/catch

Because contract violations throw catchable errors, you can use `try`/`catch` to handle them gracefully:

Listing 10.26: Catching contract violations

```
fn positive_sqrt(n: Float) -> Float
    require n >= 0.0, "cannot take sqrt of negative number"
{
    // Newton's method approximation
    flux guess = n / 2.0
    for _ in 0..20 {
        guess = (guess + n / guess) / 2.0
    }
    return guess
}

let values = [4.0, -1.0, 9.0, -16.0]

for v in values {
    let result = try {
        positive_sqrt(v)
    } catch e {
        e.get("message")
    }
    print("sqrt(${v}) = ${result}")
}
// sqrt(4.0) = 2.0
// sqrt(-1.0) = require failed in 'positive_sqrt': ...
// sqrt(9.0) = 3.0
// sqrt(-16.0) = require failed in 'positive_sqrt': ...
```

10.8 Assertions

Lattice provides a family of assertion functions for testing invariants:

Function	Purpose
`assert(cond)`	Fails if `cond` is falsy
`assert(cond, msg)`	Fails with custom message
`assert_eq(a, b)`	Fails if `a != b`
`assert_ne(a, b)`	Fails if `a == b`
`assert_true(val)`	Fails if `val` is not `true`
`assert_false(val)`	Fails if `val` is not `false`
`assert_nil(val)`	Fails if `val` is not `nil`
`assert_contains(s, sub)`	Fails if `s` does not contain `sub`
`assert_type(val, type)`	Fails if `val` is not of type `type`
`assert_throws(closure)`	Fails if closure does *not* throw

All assertion failures are catchable with `try`/`catch`. The `assert_throws()` function is particularly useful in tests—it calls a closure and succeeds only if the closure throws:

Listing 10.27: Testing that code throws

```
test "division by zero throws" {
    assert_throws(|| {
        let x = 1 / 0
    })
}

test "require contracts throw on violation" {
    fn positive(n: Int) -> Int
        require n > 0, "must be positive"
    {
        return n
    }

    assert_throws(|| { positive(-1) })
}
```

10.9 The `try_fn` Helper

The standard library provides `try_fn()`, which bridges the gap between `try`/`catch` exceptions and the `Result` type. It runs a closure inside a `try`/`catch` and returns the outcome as a Result:

Listing 10.28: try_fn converts exceptions to Results

```
let result = try_fn(|| {
    let x = 10 / 0
    x
})

print(is_err(result))         // true
print(result.get("value"))    // error map with message, line, stack
```

This is useful when you want to call a function that might throw but you prefer to work with Result maps rather than `try`/`catch` blocks:

Listing 10.29: Using try_fn with the ? operator

```
fn safe_pipeline(data: any) -> Map {
    let step1 = try_fn(|| { validate(data) })?
    let step2 = try_fn(|| { transform(step1) })?
    return ok(step2)
}
```

10.10 Designing for Graceful Failure

Now that we have seen all the error handling tools, let us discuss when to use each one.

10.10.1 Choosing the Right Tool

Situation	Recommended Approach
Expected failures (parsing, I/O, validation)	Return `Result` maps with `ok()`/`err()`
Calling code that throws	Wrap in `try`/`catch` or `try_fn()`
Chaining fallible operations	Use ? operator with Results
Bugs and impossible states	`panic()` or `assert()`
Function input validation	`require` contracts
Function output guarantees	`ensure` contracts
Resource cleanup	`defer`

10.10.2 The Error Boundary Pattern

A common architecture is to let errors propagate freely through the interior of your program and catch them at well-defined *boundaries*—the top of a request handler, the main loop of a CLI tool, the entry point of a worker:

Listing 10.30: The error boundary pattern

```
fn handle_request(req: any) -> String {
    // Interior code uses ? and Results freely
    let result = try {
        let user = authenticate(req)?
        let data = fetch_data(user)?
        let response = format_response(data)?
        unwrap(response)
    } catch e {
        // Boundary: convert any error to a user-friendly response
        let msg = e.get("message")
        print("Error handling request: ${msg}")
        "500 Internal Server Error"
    }
    return result
}
```

Interior functions return Results and use ? to propagate errors upward. The boundary catches everything, logs the details, and returns a safe response to the caller.

10.10.3 Fail Fast, Recover High

A principle that serves Lattice programs well: **fail fast at the point of failure, recover at the point where you have enough context to do something useful.**

Do not catch errors deep inside helper functions only to re-throw them. Let them propagate. Catch them where you can take meaningful action—retry, return a default, log and continue, or shut down cleanly.

Listing 10.31: Let errors propagate to where context exists

```
// Bad: catching too early
fn get_user_name(id: Int) -> String {
    let result = try {
        fetch_user(id)
    } catch e {
        // We don't know what to do here!
        return "Unknown"
    }
    return result.get("name")
}

// Better: propagate and let the caller decide
fn get_user_name(id: Int) -> Map {
    let user = fetch_user(id)?
    return ok(user.get("name"))
}

// Caller has context to handle the error
fn display_profile(id: Int) -> String {
    let name = match get_user_name(id) {
        {tag: "ok", value} => value,
        {tag: "err", value} => "Guest"
    }
    return "Welcome, ${name}!"
}
```

10.11 Exercises

1. **Safe division.** Write a function `fn safe_div(a: Float, b: Float) -> Map` that returns `ok(a / b)` when `b` is not zero and `err("division by zero")` otherwise. Then write a function `fn chain_divide(values: Array) -> Map` that divides the first element by each subsequent element in sequence, using `?` to propagate errors.

2. **Resource manager.** Write a function that simulates opening three resources (use print statements to show "opened resource N"). Use three `defer` blocks to close them. Introduce an error after opening the second resource and verify that all opened resources are still closed.

3. **Retry logic.** Write a function `fn retry(attempts: Int, action: any) -> Map` that calls the `action` closure up to `attempts` times. If the action succeeds (returns without throwing), return

`ok(result)`. If all attempts throw, return `err("all attempts failed")`. Use `try`/`catch` inside the retry loop.

4. **Contract design.** Write a function `fn clamp(value: Int, low: Int, high: Int) -> Int` with a `require` contract that ensures `low <= high`, and an `ensure` contract that verifies the result is within the `[low, high]` range. Test it with values inside and outside the range, and verify that the contracts catch violations.

5. **Pipeline with errors.** Build a data processing pipeline with three stages—validate, transform, and format—each returning a Result. Write a `fn pipeline(input: any) -> Map` that chains the three stages using `?`. Test it with inputs that fail at each stage and verify that the correct error is returned.

What's Next With control flow, functions, collections, strings, pattern matching, and error handling in your toolkit, you are equipped to write substantial Lattice programs. But we have been working with values that are either immutable by default or mutable by declaration without fully understanding *why*. In Part III, we dive deep into the *phase system*—Lattice's most distinctive feature. Chapter 11 reveals how the fluid, crystal, and sublimated phases govern mutability, how freezing and thawing work at the memory level, and why thinking of values as materials with physical states leads to safer, more predictable programs.

Part III

The Phase System

Chapter 11

Phases Explained

Every material in nature exists in a state. Water flows as liquid, hardens as ice, disperses as steam. The state determines what you can *do* with the material—you can pour water but not ice, you can stack ice but not steam. Lattice borrows this intuition and applies it to data: every value in the language carries a *phase* that governs whether it can be changed, shared, or locked down permanently.

If you have been following along from Chapter 4, you already know the basics: `flux` for mutable, `fix` for immutable, and `let` for "figure it out for me." This chapter goes deeper. We will explore the full philosophy behind the phase system, examine each phase keyword in detail, master the trio of `freeze()`, `thaw()`, and `clone()`, learn when and why to freeze values, and understand the error messages the phase checker produces when something goes wrong.

11.1 The Philosophy: Mutability as a Material Property

Most languages treat mutability as a *declaration modifier*. In Rust, you write `let mut`. In JavaScript, you choose between `let` and `const`. In both cases, the declaration tells the *binding* whether it can be reassigned, but the underlying data may or may not follow suit.

Lattice takes a different stance. Mutability is not a property of the *binding*—it is a property of the *value itself*. When you declare a variable with `flux`, the value stored inside is tagged as *fluid*: it lives in the mutable heap, the garbage collector watches over it, and you can modify it freely. When you declare with `fix`, the value is tagged as *crystal*: it is hardened, immutable, and may be stored in a memory arena where it can never be altered.

Phase

A *phase* is a runtime tag on every Lattice value that describes its mutability state. There are four phase tags in the runtime:

- **Fluid** (`VTAG_FLUID`) — mutable, alive, in motion.
- **Crystal** (`VTAG_CRYSTAL`) — immutable, hardened, permanent.
- **Unphased** (`VTAG_UNPHASED`) — newly created, not yet committed to either side.
- **Sublimated** (`VTAG_SUBLIMATED`) — sealed against in-place mutation and against channel sends. Not un-thawable, though: see the warnbox in Section 11.6.4.

This design has a profound consequence: you can take a fluid value and *freeze* it into a crystal. You can take a crystal and *thaw* it back into a fluid. You can *clone* a value to get an independent copy without changing the original's phase. The phase is the material property of the data itself, and Lattice gives you the tools to change that material state deliberately.

Why does this matter? Consider a configuration object. While your application is starting up, the configuration needs to be mutable—you are loading defaults, overriding with environment variables, parsing command-line flags. But once startup is complete, the configuration should be frozen solid: no accidental mutation, no race conditions, no surprises. In most languages, you enforce this with discipline and documentation. In Lattice, you enforce it with a single call to `freeze()`.

Listing 11.1: Freezing a configuration after setup

```
flux config = Map::new()
config.set("host", "0.0.0.0")
config.set("port", "8080")
config.set("debug", "false")

// Startup complete. Lock it down.
fix app_config = freeze(config)
print(phase_of(app_config))  // "crystal"
```

After the freeze, `app_config` is crystal. Calling `app_config.set("host", "evil.com")` stops the program:

Listing 11.2: A mutation on hardened material

```
vm error: cannot call mutating method 'set' on crystal value
          'app_config' (use thaw(app_config) to make it mutable)
```

The data is safe—not because of a linting rule or a code review comment, but because the material itself has hardened.

Reading Error Messages in This Book

Every backend reports the same core message; only the prefix differs. The default stack VM writes `vm error:`, the tree-walking interpreter (`--tree-walk`) writes `error:`, and the register VM (`--regvm`) writes `regvm error:`. Errors raised by the static phase checker—the strict-mode diagnostics later in this chapter—carry `phase error:` on all three. When we quote a message from here on, we quote the core text and leave the prefix implied.

11.2 `flux` (Fluid) — Mutable, Alive, in Motion

The `flux` keyword declares a binding whose value is in the *fluid* phase. Fluid values can be reassigned, mutated in place, grown, shrunk, and generally treated as living, changing data.

Listing 11.3: Basic `flux` usage

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

When you create a value with `flux`, the runtime tags it as `VTAG_FLUID`. Under the hood (in `include/value.h`), every `LatValue` struct carries a `phase` field of type `PhaseTag`. A fluid value's phase tag is set to `VTAG_FLUID`, and it lives on the *FluidHeap*—the garbage-collected portion of memory managed by the mark-sweep collector.

11.2.1 Fluid Collections

Fluid values are not limited to scalars. Arrays, maps, sets, and structs can all be fluid:

Listing 11.4: Fluid collections

```
flux temperatures = [18.5, 19.2, 20.1]
temperatures.push(21.0)
temperatures.push(22.3)
print(temperatures)  // [18.5, 19.2, 20.1, 21, 22.3]

flux users = Map::new()
users.set("alice", 42)
users.set("bob", 37)
print(users.len())  // 2
```

When a compound value like an array is fluid, all of its elements inherit the fluid phase. You can modify the array and its contents freely.

11.2.2 Fluid Reassignment

Because `flux` values are mutable bindings, you can reassign them entirely:

Listing 11.5: Reassigning a `flux` binding

```
flux greeting = "hello"
print(phase_of(greeting))  // "fluid"

greeting = "bonjour"
print(greeting)  // "bonjour"

// Even change the type entirely
greeting = 42
print(greeting)  // 42
```

Lattice is dynamically typed, so a `flux` binding can hold any value. The phase stays fluid unless you explicitly transition it.

Fluid Does Not Mean Unprotected

A fluid value is mutable, but it is still tracked by the runtime. The garbage collector manages its memory, the phase checker watches for incorrect transitions, and in strict mode, the compiler verifies that you do not accidentally pass a fluid value where a crystal is expected.

11.3 fix (Crystal) — Immutable, Hardened, Permanent

The `fix` keyword declares a binding whose value is in the *crystal* phase. Crystal values are immutable: their internal state cannot be modified, and any attempt to mutate them in place produces a runtime error. (The *binding* is a separate matter, and a subtle one—we come back to it in Section 11.3.3.)

Listing 11.6: Basic `fix` usage

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

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

`fix` Already Crystallizes

A `fix` binding hardens whatever it binds, and it hardens all the way down. `fix a = [[1], [2]]` gives you a crystal array of crystal arrays—`phase_of(a[0])` is `"crystal"`, and `a[0].push(9)` errors—with no `freeze()` in sight. The explicit `freeze()` in the listing above is there for emphasis and for the reader; you reach for it in earnest when the value is already living under some *other* name and you want to harden it in place (Section 11.5).

11.3.1 Why Crystal Is More Than "const"

In many languages, `const` merely prevents *rebinding*—you cannot assign a new value to the variable, but you can still modify the data it points to. JavaScript's `const` is the canonical example:

Listing 11.7: The problem with shallow const (JavaScript-style)

```
// In JavaScript:
// const arr = [1, 2, 3];
// arr.push(4);  // This works! const doesn't protect contents.

// In Lattice:
fix arr = freeze([1, 2, 3])
// arr.push(4)
// error: cannot call mutating method 'push' on crystal
//        value 'arr' (use thaw(arr) to make it mutable)
```

Lattice's crystal phase is *deep*. When you freeze an array, every element becomes crystal. When you freeze a struct, every field becomes crystal. When you freeze a map, every key-value pair becomes crystal. The immutability propagates through the entire object graph.

Under the hood, the `value_freeze()` function in `src/value.c` calls `set_phase_recursive()`, which walks the entire value tree and sets every nested value's phase to `VTAG_CRYSTAL`:

Listing 11.8: Deep freeze in action

```
flux nested = [[1, 2], [3, 4], [5, 6]]
fix frozen = freeze(nested)

print(phase_of(frozen))     // "crystal"
print(phase_of(frozen[0]))  // "crystal"

// frozen[0].push(9)
// error: cannot call mutating method 'push' on a frozen value
```

Note the wording of that second message. When the receiver is a bare variable, Lattice names it and suggests the remedy; when the receiver is an element reached through an index or field chain, there is no name to print, so the message shortens to `on a frozen value`. Same rule, same enforcement—less to say.

11.3.2 Crystal Values and Memory

Crystal values may be relocated from the FluidHeap into a *CrystalRegion*—an arena-backed memory area that provides cache locality and O(1) bulk deallocation. We explore this memory architecture in detail in Chapter 12. For now, the key insight is that crystal values are not merely "values you are not allowed to change." They are values that *the runtime treats differently at the memory level*, storing them in a region optimized for immutable data.

11.3.3 Freezing Protects the Value, Not the Name

Here is the corner that trips up nearly everyone who arrives from Rust or from a `const`-heavy language. A phase is a property of a *value*. The name you happen to have pointed at that value is not part of the crystal.

Listing 11.9: A `fix` binding can be pointed somewhere else

```
fix x = 1
x = 5
print(x)  // 5  -- no error

fix a = [1]
a = [2, 2]
print(phase_of(a))  // "unphased"
```

Rebinding does not thaw anything. It discards the old value entirely—crystal and all—and installs a new one, which arrives with whatever phase it was born with. The array literal `[2, 2]` is unphased, so that is what a now reports.

What rebinding cannot do is reach *into* a crystal:

Listing 11.10: In-place mutation is still blocked

```
fix a = [1]
// a.push(2)
// error: cannot call mutating method 'push' on crystal
//        value 'a' (use thaw(a) to make it mutable)
```

So `fix` is a statement about the material, not a lock on the label. If you want the label locked too, that is exactly what strict mode buys you—see Section 11.9, where `fix count = 42; count = 43` becomes a compile-time refusal.

Compound Assignment on a Frozen Local Diverges

`+=` against a frozen value is a mutation, not a rebind, and the backends do not agree on where they catch it. Inside a function body, `fix y = 1; y += 1` raises `cannot modify a frozen value` on the default stack VM and on `--regvm`, but the tree-walking interpreter (`--tree-walk`) currently allows it and prints `2`. At global scope, `fix x = 1; x += 1` is allowed on all three. This is a known divergence in v0.6.4; do not build a program that depends on either answer.

11.4 `let` — Phase Inference

Not every binding needs an explicit phase declaration. The `let` keyword tells Lattice: "I do not care about the phase right now—infer it from context."

Listing 11.11: `let` bindings with inferred phase

```
let name = "Lattice"
let count = 42
let items = [1, 2, 3]

print(phase_of(name))   // "unphased"
print(phase_of(count))  // "unphased"
print(phase_of(items))  // "unphased"
```

When you use `let`, the value starts in the `VTAG_UNPHASED` state. Unphased values are not locked into either fluid or crystal. They behave as fluid in practice—you can reassign and mutate them—but they do not carry the explicit guarantee of either phase.

Unphased

An *unphased* value (`VTAG_UNPHASED`) has no explicit phase commitment. It can be treated as mutable and can transition to either fluid or crystal through explicit operations. In casual mode (the default), `let` bindings are unphased. In strict mode, `let` is *forbidden*—you must choose `flux` or `fix` explicitly.

The phase checker in `src/phase_check.c` handles `let` differently depending on the execution mode. In casual mode (`MODE_CASUAL`), the effective phase of a `let` binding is inferred from the initializer expression:

Listing 11.12: Phase inference with `let`

```
let frozen_data = freeze([1, 2, 3])
// frozen_data's effective phase is crystal (inferred from freeze())

let thawed_copy = thaw(frozen_data)
// thawed_copy's effective phase is fluid (inferred from thaw())
```

When to Use `let`

Use `let` for quick prototyping and REPL exploration. When writing production code, prefer `flux` or `fix` to make your intent explicit. In strict mode, you are *required* to make this choice—the compiler will refuse `let` bindings entirely.

11.4.1 The Relationship Between `let` and Strict Mode

In strict mode (`#mode strict`), the phase checker rejects `let` bindings with a clear error:

Listing 11.13: `let` rejected in strict mode

```
#mode strict

let name = "Lattice"
// Error: strict mode: use 'flux' or 'fix' instead of 'let'
//        for binding 'name'
```

This is by design. Strict mode exists for codebases where phase discipline matters—library code, concurrent systems, safety-critical paths. We cover strict mode in full detail in Chapter 14.

11.5 `freeze()`, `thaw()`, `clone()` in Depth

These three built-in functions are the tools you use to change a value's phase or create independent copies. Think of them as the laboratory equipment for Lattice's material science.

11.5.1 `freeze()` — Crystallization

`freeze()` takes a value and hands back a crystal one, deeply immutable all the way down.

As of Lattice v0.5.0, **immutability is enforced on every backend**. Every index assignment, field assignment, and mutating method call on a frozen value raises a runtime error—there is no way to silently mutate frozen data.

There is one wrinkle worth learning before anything else, because it changes how the rest of this section reads. When you hand `freeze()` a bare variable name, it crystallizes *that variable too*:

Listing 11.14: `freeze()` on a name hardens the name

```
flux temperatures = [20.1, 21.5, 19.8, 22.0]
fix snapshot = freeze(temperatures)

print(phase_of(snapshot))      // "crystal"
print(phase_of(temperatures))  // "crystal" -- frozen in place too
```

This is deliberate. The overwhelmingly common intent behind `freeze(config)` is "this thing is done changing now," and leaving a still-molten alias behind would defeat the point. So the runtime writes

the crystallized value back into the binding. You can see the machinery on the compiled backends in `src/stackcompiler.c`, which emits a duplicate-and-store after the freeze whenever the argument is a plain identifier; the tree-walker does the same thing directly in `src/eval.c`, where the casual-mode path is commented "freeze the binding in-place."

If what you actually wanted was a frozen *snapshot* beside a still-fluid original, freeze a copy—any expression that is not a bare name is left alone:

Listing 11.15: Freezing a copy leaves the original molten

```
flux temperatures = [20.1, 21.5, 19.8, 22.0]
fix snapshot = freeze(clone(temperatures))

print(phase_of(temperatures))  // "fluid"
print(phase_of(snapshot))      // "crystal"

// The fluid value is still alive and mutable
temperatures.push(23.1)
print(temperatures)  // [20.1, 21.5, 19.8, 22, 23.1]

// The crystal snapshot is frozen in time
print(snapshot)  // [20.1, 21.5, 19.8, 22]
```

The Rule in One Line

`freeze(name)` hardens the value *and* the name. `freeze(clone(name))` hardens only the copy. The same rule governs `thaw()`—see Section 11.5.2.

Under the hood, `value_freeze_to_region()` in `src/value.c` runs a short decision procedure:

1. **Scan for shareability.** `value_is_shareable()` walks the value looking for closures, `Ref` cells, iterators, channels, or sublimated members. Any of those, and the value takes the plain path in step 4.
2. **Check that regionizing is worth it.** Scalars never regionize; strings do only at 32 bytes or more (`REGION_SHARE_MIN_STR_LEN`).
3. **Materialize.** A sealed, reference-counted crystal region is created, the value is force-copied into its arena pages, and `region_tag_recursive()` stamps `VTAG_CRYSTAL` plus the region handle onto every node in the tree.
4. **Or fall back to a tag flip.** Values that fail step 1 or step 2 stay on the ordinary heap and get their phase tags set to `VTAG_CRYSTAL` recursively.

A materialized crystal has no pointers back into the FluidHeap at all, which is what lets the collector skip it wholesale. We take the memory side of this apart in Chapter 12; the behavioural side—why the difference is invisible to your program—is Section 11.8.

Freeze Is Not Free

`freeze()` copies. For large nested structures, this has a real cost in both time and memory—roughly $1.9\times$ a bare tag flip for a value that materializes into a region. What the cost buys is a value that is truly independent, provably immutable, and from that moment on free to alias: every later copy, argument pass, and channel send of it is O(1). Freeze once, pass freely.

Freezing with Contracts

`freeze()` accepts an optional `where` clause—a closure that receives the value and gets a chance to object before crystallization:

Listing 11.16: Freeze with contract validation

```
flux score = 85

fix validated = freeze(score) where |val| {
    assert(val >= 0, "score must be non-negative")
    assert(val <= 100, "score must not exceed 100")
    return val
}

print(validated)             // 85
print(phase_of(validated))   // "crystal"
```

Note the shape: `where` follows the closing parenthesis, and the closure is not an argument. Passing the closure inside the parentheses, comma and all, is a parse error.

The contract's *return value is ignored*. What matters is whether it raises: if the closure fails—an `assert` that does not hold, a division by zero, anything—the freeze aborts and the failure is wrapped:

Listing 11.17: A contract that refuses

```
flux score = 150

fix validated = freeze(score) where |val| {
    assert(val <= 100, "score must not exceed 100")
    return val
}
// error: freeze contract failed: ...
//        score must not exceed 100
```

This lets you enforce invariants at the moment of crystallization—the value only hardens if it passes.

Returning a Value Is Not Refusing

A contract that *returns* a complaint—a string, `false`, `nil`—does not stop the freeze. A contract body of `return val <= 100` crystallizes a score of 150 without a murmur. Contracts must raise to refuse; reach for `assert()`.

11.5.2 `thaw()` — Melting a Crystal

`thaw()` is the inverse of `freeze()`. It takes a crystal value and returns a *new* fluid copy—never a mutable window onto the original crystal's storage.

Listing 11.18: The `thaw()` function

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

flux thawed = thaw(frozen_list)
print(phase_of(thawed))  // "fluid"

// Now we can modify the thawed copy
thawed.push(40)
print(thawed)       // [10, 20, 30, 40]
print(frozen_list)  // [10, 20, 30]  -- data unchanged
```

The implementation in `src/value.c` spells out the guarantee: `value_thaw()` takes a private force-copy via `value_copy_out()`, rebuilds any crystallized map layout with `value_decrystallize()`, then calls `set_phase_recursive()` to mark the copy `VTAG_FLUID`. The comment there is emphatic—shared

region memory, phase tags included, is never written, so every other alias on every other thread observes nothing at all.

But look again at that listing. `frozen_list` still holds `[10, 20, 30]`, and yet:

Listing 11.19: `thaw()` melts the name it was given

```
fix frozen_list = freeze([10, 20, 30])
flux thawed = thaw(frozen_list)

print(phase_of(frozen_list))  // "fluid"  -- melted in place
frozen_list.push(99)
print(frozen_list)  // [10, 20, 30, 99]
print(thawed)       // [10, 20, 30]   -- a separate copy
```

Symmetry with `freeze()`: when the argument is a bare identifier, the new phase is written back into the binding. The *data* is copied—`thawed` and `frozen_list` are now two independent fluid arrays—but the name you thawed does not stay crystal.

Thaw Copies the Data and Melts the Name

Two things happen at once, and it is worth keeping them apart. `thaw(x)` returns a fresh, independent fluid copy—that part is unconditional, and it is why other aliases of the frozen value are never disturbed. Additionally, when `x` is a plain variable, `x` itself becomes fluid. Wrap the argument (`thaw(clone(x))`) when you need `x` to stay hardened.

Thawing Refs

When `thaw()` operates on a `Ref` value (a reference-counted shared wrapper), it breaks the sharing. The thawed result is a new `Ref` with a deep-cloned inner value, independent of any other references to the original. The clone happens under the cell's lock, so a sibling thread cannot pull the value out from under it:

Listing 11.20: Thawing a Ref breaks sharing

```
fix shared_ref = freeze(Ref::new(42))
print(phase_of(shared_ref))  // "crystal"

flux independent = thaw(shared_ref)
print(phase_of(independent))  // "fluid"
print(independent.get())      // 42
// independent is a new Ref, no longer sharing data with shared_ref
```

Constructing Refs

The constructor is `Ref::new(v)`. There is no bare `Ref(v)` form—it fails with `undefined variable 'Ref'`.

Mutating Through a Ref: Copy-on-Write

A `Ref` is the one shared *mutable* cell in the language, so what happens when you put a frozen value inside a fluid one and then try to mutate it?

Listing 11.21: A fluid Ref over a crystal inner value

```
let inner = freeze([1, 2])
flux r = Ref::new(inner)

r.push(3)                 // no error

print(r.get().len())      // 3
print(inner.len())        // 2   -- the original alias is untouched
print(phase_of(r.get()))  // "crystal"
```

The mutation succeeds, and it succeeds by *copying*. The cell privatizes its frozen contents before writing, so the alias you froze keeps its two elements while the cell moves on to three. The result stays crystal-phased: you have a new frozen value in the cell, not a thawed one.

This behaves identically on all three backends as of v0.6.4—earlier tree-walker builds raised an error here instead. Direct mutation of a frozen value that is *not* behind a fluid `Ref` still errors, exactly as Section 11.5.1 describes. The Ref is the escape hatch, and copy-on-write is the price of using it.

11.5.3 clone() — Independent Copy

clone() creates a deep copy of a value *without changing its phase*. A fluid value cloned stays fluid. A crystal value cloned stays crystal. The result is a structurally identical but completely independent copy.

Listing 11.22: The clone() function

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

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

print(phase_of(original))  // "fluid"
print(phase_of(copy))      // "fluid"  -- same phase as original
```

Clone is especially useful when you want to pass a value to a function that might modify it, but you want to preserve the original:

Listing 11.23: Cloning for safe function calls

```
fn process_data(data: any) {
    // This function modifies its argument
    data.push(0)
    return data
}

flux sensor_readings = [45.2, 47.8, 46.1]
flux processed = process_data(clone(sensor_readings))

print(sensor_readings)  // [45.2, 47.8, 46.1]  -- safe
print(processed)        // [45.2, 47.8, 46.1, 0]
```

When to Use Each

- Use freeze() when you want to lock a value down for safety, sharing, or performance.
- Use thaw() when you need to modify data that is currently frozen.
- Use clone() when you want an independent copy without changing the phase.

11.6 Advanced Phase Operations

Beyond the fundamental trio, Lattice offers several advanced phase operations for fine-grained control over the crystallization lifecycle.

11.6.1 `forge` — Scoped Construction

A `forge` block lets you construct a complex crystal value through a series of mutable operations, automatically freezing the result when the block completes. Think of it like a foundry: the material is molten inside the forge, but what comes out is solid.

Listing 11.24: Building a crystal value with `forge`

```
fix server_config = forge {
    flux temp = Map::new()
    temp.set("host", "localhost")
    temp.set("port", "8080")
    temp.set("max_connections", "1000")
    temp.set("timeout_ms", "30000")
    freeze(temp)
}

print(phase_of(server_config))  // "crystal"
print(server_config.get("host"))  // "localhost"
```

The `forge` block is especially valuable when the construction process requires loops, conditionals, or function calls:

Listing 11.25: Complex construction inside `forge`

```
fix lookup_table = forge {
    flux table = Map::new()
    for i in 0..100 {
        table.set(to_string(i), i * i)
    }
    freeze(table)
}

print(lookup_table.get("7"))   // 49
print(lookup_table.get("12"))  // 144
```

The phase checker knows that a `forge` block always produces a crystal result. This is reflected in `src/phase_check.c`, where `EXPR_FORGE` returns `PHASE_CRYSTAL`.

11.6.2 `anneal` — Transform a Crystal

In metallurgy, *annealing* is the process of heating a metal, transforming it, and letting it cool back to a hardened state. Lattice's `anneal` does the same: it thaws a crystal value, applies a transformation closure, and re-freezes the result.

Listing 11.26: Annealing a crystal value

```
fix prices = freeze([9.99, 14.99, 19.99, 24.99])

// Apply a 10% discount without manually thawing/re-freezing
fix discounted = anneal(prices) |items| {
    flux result = []
    for price in items {
        result.push(price * 0.9)
    }
    return result
}

print(discounted)              // [8.991, 13.491, 17.991, 22.491]
print(phase_of(discounted))    // "crystal"
```

The closure follows the parentheses, the way `where` does on `freeze()`—it is not an argument. Passing it as one earns the parser's usual objection. The transformation closure receives the thawed (fluid) copy, and whatever it returns is automatically frozen.

Do Not Read the Target After Annealing

Whether `anneal(x)` leaves `x` holding the original crystal or the transformed one is not settled across backends in v0.6.4: the default stack VM and `--tree-walk` write the result back into `x`, while `--regvm` leaves it alone. Bind the result and use the binding.

11.6.3 `crystallize` and `borrow` — Scoped Phase Changes

Sometimes you want to temporarily change a value's phase within a specific scope. `crystallize` temporarily freezes a variable for the duration of a block, then restores its original phase:

Listing 11.27: Temporary crystallization

```
flux data = [1, 2, 3]

crystallize(data) {
    // Inside this block, data is crystal
    print(phase_of(data))  // "crystal"
    // data.push(4)  // This would error!
}

// Back to fluid outside
print(phase_of(data))  // "fluid"
data.push(4)
print(data)  // [1, 2, 3, 4]
```

`borrow` is the inverse: it temporarily thaws a crystal variable so you can modify it within a block, then re-freezes it:

Listing 11.28: Temporarily borrowing a crystal value

```
fix config = freeze(Map::new())

borrow(config) {
    // Inside this block, config is fluid
    config.set("debug", "true")
    config.set("verbose", "true")
}

// config is crystal again outside the borrow block
print(phase_of(config))  // "crystal"
```

These scoped operations are compiled to a sequence of phase tag changes and are verified by the phase checker (see `src/stackcompiler.c`). The key property: both `crystallize` and `borrow` restore the original phase when the block ends, even if an error occurs during execution.

11.6.4 `sublimate` — The Strictest Lock

In chemistry, sublimation is when a solid transforms directly into gas, skipping the liquid state entirely. In Lattice, `sublimate` converts a value into the *sublimated* phase (`VTAG_SUBLIMATED`)—the strictest in-place immutability the runtime offers. No mutation reaches a sublimated value, and no channel will carry one.

Listing 11.29: Sublimating a value

```
flux secret_key = "my-api-key-12345"
sublimate(secret_key)

print(phase_of(secret_key))  // "sublimated"
```

Sublimated values get their own error vocabulary, distinct from crystal's. Where a frozen value says *frozen* and points you at `thaw()`, a sublimated value says no:

Listing 11.30: What sublimation forbids

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

// data[0] = 9
// error: cannot modify a sublimated value

// data.push(4)
// error: cannot call mutating method 'push' on a sublimated value

// ch.send(data)
// error: cannot send a sublimated value on a channel
```

Note that last one. Crystal values are the *preferred* channel payload; sublimated values are rejected outright, and so is any value that merely contains one. Sublimation says "this stays here," which makes it a fit for security-sensitive material (API keys, credentials, certificates) that you never want copied into a worker thread by accident.

Sublimation Guards the Value, Not the Data

The guarantee sublimation offers is precise: nothing mutates this value in place, and nothing sends it over a channel. It is not a vault, and it is not a one-way door. `thaw()` still hands back a fluid copy of a sublimated value in v0.6.4—and, following the write-back rule from Section 11.5.2, `thaw(secret)` on a bare name leaves `secret` itself fluid again:

```
flux secret = [1, 2, 3]
sublimate(secret)

// An rvalue argument leaves the seal on secret intact.
flux guarded = thaw(clone(secret))
guarded.push(4)
print(guarded)           // [1, 2, 3, 4]
print(phase_of(secret))  // "sublimated"

// A bare name writes the new phase back into the binding.
flux melted = thaw(secret)
print(phase_of(melted))  // "fluid"
print(phase_of(secret))  // "fluid" -- secret itself is molten again
```

Identical on all three backends. Treat sublimation as the strongest *in-place* guarantee in the language, and do not lean on it as a secrecy mechanism.

11.6.5 `freeze(x) except [...]` — Partial Freezing

Sometimes you want to freeze most of a container but leave a few keys or fields writable. `freeze(x) except ["k1", "k2"]` does exactly that:

Listing 11.31: Freezing a map with exempt keys

```
flux m = Map::new()
m["host"] = "example.com"
m["retries"] = 0
m["timeout"] = 30

freeze(m) except ["retries"]

// The map itself reports crystal
print(phase_of(m))  // "crystal"

m["retries"] = 5           // OK  -- exempt key stays writable
print(m["retries"])        // 5
m.set("retries", 7)        // OK  -- so does the method form
print(m["retries"])        // 7
m["retries"] += 1          // OK
print(m["retries"])        // 8
m.remove("retries")        // OK
print(m.len())             // 2

// m["host"] = "x"
// error: cannot modify a frozen value
```

Think of it as a crystal with deliberate defects in the lattice: the bulk of the structure is rigid, and a named handful of sites stay mobile. Four rules govern the behaviour, identically on all three backends:

1. **The container reports crystal.** `phase_of(m)` returns `"crystal"` after `freeze(m) except [...]`, even though some keys are exempt. The exemptions are per-key overrides recorded alongside the map, not a different phase for the map as a whole.

2. **An exempt key is fully writable.** Assignment (`m["k"] = v`), compound assignment (`m["k"] += 1`), and the mutating methods that target that one key—`m.set("k", v)` and `m.remove("k")`—all succeed. Lattice v0.5.0 blocked the method forms; v0.6.4 relaxed that, so the two spellings now agree. Whole-container mutators are still refused, because they cannot promise to touch only exempt keys: `m.merge(other)` raises `cannot call mutating method 'merge' on crystal value 'm'`.

3. **A later full freeze closes all exemptions.** `freeze(m) except ["retries"]` followed later by a plain `freeze(m)` locks every key, previously exempt ones included. A `thaw()` likewise drops all per-key overrides.

4. **New keys are never exempt.** A key that did not exist when `freeze(...) except` ran cannot be created afterwards, even if its name appears in the exemption list. The exemption marks existing sites as mobile; it does not license growth.

Listing 11.32: Refreezing closes exemptions

```
flux m = Map::new()
m["host"] = "h"
m["retries"] = 0

freeze(m) except ["retries"]
m["retries"] = 5      // OK
print(m["retries"])   // 5

freeze(m)             // full refreeze
// m["retries"] = 6
// error: cannot modify a frozen value
```

Why Refreezing Seals the Holes

An exempt map is deliberately kept out of shared crystal storage—the per-key override table (`key_phases` in `src/value.c`) has to stay writable, and sealed regions do not do writable. When a plain `freeze()` comes along, the value is materialized properly, and `region_tag_recursive()` drops that table on the floor: with no overrides left, every key inherits the map's crystal phase. The holes do not close because a rule says so. They close because the record of where they were is gone.

11.7 When and Why to Freeze Values

Knowing *how* to freeze is only half the story. Knowing *when* to freeze is what separates a novice Lattice programmer from an expert. Here are the primary scenarios where freezing pays dividends.

11.7.1 Sharing Data Between Threads

In Lattice's structured concurrency model (covered in Chapter 19), spawned tasks should not share mutable state. The strict mode phase checker enforces this: fluid bindings cannot cross a spawn boundary. The solution is to freeze data before sharing:

Listing 11.33: Freezing data for safe sharing across tasks

```
flux results = [10, 20, 30, 40, 50]
fix shared = freeze(results)

scope {
    spawn {
        // Safe: shared is crystal, no data races
        print("Task sees: " + to_string(shared))
    }
    spawn {
        print("Other task sees: " + to_string(shared))
    }
}
```

Crystal values are inherently safe to share because no one can modify them. This eliminates an entire class of concurrency bugs—no locks needed, no synchronization overhead, no data races.

With crystal sharing (Section 11.8), spawning a thread that reads frozen data from the enclosing scope is O(1)—the frozen value is shared by reference count instead of deep-copied.

11.7.2 API Boundaries

When designing a function that returns data the caller should not modify, freeze it:

Listing 11.34: Returning frozen data from a function

```
fn get_default_settings() {
    flux defaults = Map::new()
    defaults.set("theme", "dark")
    defaults.set("font_size", "14")
    defaults.set("language", "en")
    return freeze(defaults)
}

fix settings = get_default_settings()
// Callers get crystal data---they can read but not modify
```

11.7.3 Snapshot Semantics

When you need to capture the state of a value at a particular moment, freeze it:

Listing 11.35: Creating snapshots with freeze

```
flux log_entries = []
log_entries.push("Server started")
log_entries.push("Listening on port 8080")

fix startup_log = freeze(clone(log_entries))

log_entries.push("Connection from 192.168.1.1")
log_entries.push("Request: GET /api/status")

fix runtime_log = freeze(clone(log_entries))

print(startup_log.len())  // 2
print(runtime_log.len())  // 4
```

Each `freeze()` call captures the array at that point in time. The snapshots are independent and immutable, while the original continues to grow.

The `clone()` is doing real work here, and it is the write-back rule from Section 11.5.1 showing up in anger. Write `freeze(log_entries)` instead and the very next `push` stops the program—you would have frozen the log itself, not a picture of it. Snapshots are the one pattern where you almost always want `freeze(clone(x))`.

11.7.4 Performance Optimization

Crystal values stored in arena-backed CrystalRegions enjoy better cache locality than fluid values scattered across the general heap. For read-heavy workloads with large data structures, freezing the data can improve access performance.

With crystal sharing (Section 11.8), passing or copying frozen data is O(1)—the value is shared by reference count rather than deep-copied. This makes the freeze-before-sharing pattern not only correct but also the high-performance choice.

11.7.5 Correctness Guarantees

Sometimes you freeze a value only to catch bugs. If you know a value should not change after a certain point, freezing it converts silent corruption into a loud runtime error:

Listing 11.36: Freezing for correctness

```
fn validate_and_seal(data: any) {
    assert(data.len() > 0, "record must not be empty")
    return freeze(data)
}

flux raw_input = ["alice", "admin"]
fix user_record = validate_and_seal(raw_input)

print(phase_of(user_record))  // "crystal"
// Any code that accidentally tries to mutate user_record
// gets an immediate phase error instead of silent corruption
```

11.8 Crystal Sharing — How Frozen Values Are Stored

As of Lattice v0.6.0, freezing a container stores it **once**, and every subsequent copy shares it by reference with reference counting. Assigning to a variable, passing to a function, returning from a function, sending over a channel, and spawning a thread over frozen data are all O(1) regardless of the value's size. Before v0.6.0, Lattice deep-copied the frozen value in every one of those situations.

Semantics Did Not Change

Lattice is still strictly pass-by-value from the program's point of view. Sharing is an optimization made safe by the v0.5.0 enforcement work: a frozen value provably cannot be mutated, so aliasing it is unobservable. A differential test mode that disables all sharing produces bit-identical program output.

11.8.1 What Shares and What Does Not

Two kinds of frozen value exist internally:

- **Shared:** frozen arrays, maps, structs, sets, tuples, buffers, enums, and strings $\geq$ 32 bytes — stored once in shared immutable storage.
- **Legacy (plain copy):** values containing closures, Refs, iterators, or channels anywhere in their graph; values with sublimated members; scalars; strings under 32 bytes. These freeze exactly as before—an immutability tag with ordinary copying.

A program **cannot observe** which kind it has—behaviour is identical; only performance differs.

11.8.2 How Operations Behave Since v0.6.0

Operation	Behaviour
`freeze(container)`	One-time copy into shared storage (about 1.9× the cost of a plain pre-v0.6.0 freeze, paid once).
copy / pass / return / `send` / `spawn` over frozen data	O(1) share (reference count increment).
`thaw(v)`	Always a fresh mutable copy—other aliases of the frozen value are unaffected.
`clone(v)`	Always a guaranteed physical copy. See note below.

Table 11.1: Crystal sharing operation costs.

When to Use `clone()` on Frozen Data

Extracting one element from a large shared frozen structure yields a handle that keeps the entire structure's memory alive. Use `clone(elem)` to cut the tie and release the rest.

11.8.3 Concurrency Interaction

- `ch.send(frozen_value)` is now a reference-count increment plus a tiny handle enqueue—previously two full deep copies.

- `spawn` blocks that read frozen data from the enclosing scope share it instead of deep-copying the environment.

- All send eligibility rules (Section 20.1.2) are checked first; sharing replaces the copy only for values that were already legal to send.

Listing 11.37: Spawning over frozen data shares, doesn't copy

```
fn build_huge_array() {
    flux out = []
    for i in 0..100000 { out.push(i * i) }
    return out
}

fn consume(rows: any) { print(rows.len()) }

fix dataset = freeze(build_huge_array())
scope {
    spawn { consume(dataset) }   // shares the frozen data --- no copy
    spawn { consume(dataset) }
}
```

11.8.4 Switches and Debugging Modes

Three controls let you disable or verify crystal sharing:

- `clat -no-regions`: disables sharing wholesale; every freeze behaves like a plain pre-v0.6.0 tag flip. Inherited by spawned threads.
- `LATTICE_SHARE_CRYSTALS=0` (environment variable): runtime kill switch with the same effect as `-no-regions`; works for embedded hosts and the `clat-run` thin runtime where there is no CLI flag. Any value other than exactly `0` leaves sharing on.
- `LATTICE_FORCE_COPY=1` (environment variable): debugging oracle — every would-be share becomes a deep copy. Program output must be identical with and without it; if it differs, that is a bug to report.

11.8.5 Performance

Measured on an Apple M3 Max:[1]

- Aliasing a frozen 10k-element array 2,000×: ≥1,250× faster than copying on both VMs; 6.2× on the tree-walker.
- Channel streaming of frozen payloads: 9.2–14.5× faster.
- Passing a frozen map as a function argument 100k×: 4.8–11.4× faster (per backend).

[1] Full benchmark tables in the Lattice repository at `benchmarks/RESULTS.md`; regenerate with `make bench-cbr`.

- Plain reads of frozen data: unchanged (reads were never copies).

The cost case: a loop that clones and freezes a fresh 10k array each iteration runs about 1.9× slower than with sharing disabled—the one-time materialization cost. Freeze-heavy code that never aliases can use `-no-regions`.

11.8.6 Known Limitation

Two `Ref` cells that mutually contain each other, thawed concurrently from different threads, can deadlock (per-cell locking; requires deliberately constructed mutual ref cycles). This is a documented limitation, not a planned behaviour.

11.9 Phase Errors and What They Tell You

When you violate the phase system's rules, Lattice produces clear error messages. Understanding these errors is essential for working effectively with phases.

They come from two places, and it pays to know which. *Runtime* phase errors—the `vm error:` / `error:` / `regvm error:` family—fire when a mutation actually reaches a frozen value. *Static* phase errors carry the `phase error:` prefix and come from the phase checker in `src/phase_check.c`, which runs before a single instruction executes. Almost everything in strict mode is the second kind.

11.9.1 Mutating a Crystal Value

The most common phase error: trying to modify something that is frozen.

Listing 11.38: Crystal mutation errors

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

// arr.push(4)
// error: cannot call mutating method 'push' on crystal value
//        'arr' (use thaw(arr) to make it mutable)

// arr[0] = 99
// error: cannot modify a frozen value
```

Two messages, two paths. Assignment through an index or a field always reports `cannot modify a frozen value`. A mutating *method* call gets the longer message, complete with the variable's name and the remedy, because the runtime knows which method you called and on what.

Which methods count as mutating is not a judgement call—there is a classifier, `builtin_method_mutates()` in `src/builtin_methods.c`, shared by all three backends, and it lists them: `Array.push`, `Array.pop`, `Array.insert`, `Array.remove_at`; `Map.set`, `Map.remove`, `Map.merge`; `Set.add`, `Set.remove`, `Set.clear`; and on `Buffer`, `push`, `push_u16`, `push_u32`, every `write_*` method, `clear`, `fill`, and `resize`. Everything else reads—including the methods that look destructive but return copies, like `Array.sort`, `Array.reverse`, and `Set.union`. `len`, `read_u8`, `get`, `keys`, `contains`, `slice`, indexing—none of these are affected by phase. Reading frozen data always works, and it is the whole point of freezing it.

Since v0.5.0 this holds on every backend, and since v0.6.4 it holds for frozen temporaries too, not only for named variables. The fix is to either `thaw()` the value first or restructure your code so the mutation happens before the freeze.

11.9.2 Assigning to a Crystal Binding

In casual mode—the default—there is no such error. Rebinding a `fix` name is permitted, as Section 11.3.3 showed. Strict mode is where the name gets locked down too:

Listing 11.39: Crystal assignment error in strict mode

```
#mode strict

fix count = 42
// count = 43
// phase error: strict mode: cannot assign to crystal
//              binding 'count'
```

If you need a new value, create a new binding:

Listing 11.40: Working with crystal bindings

```
fix count = 42
fix new_count = count + 1
print(new_count)  // 43
```

11.9.3 Strict Mode: `let` Not Allowed

Listing 11.41: Strict mode rejects `let`

```
#mode strict

// let x = 10
// phase error: strict mode: use 'flux' or 'fix' instead
//              of 'let' for binding 'x'

flux x = 10  // Use this instead
```

In strict mode, every binding must have an explicit phase. This forces you to think about mutability at every declaration point.

11.9.4 Strict Mode: Phase Mismatch

Listing 11.42: Strict mode phase mismatch

```
#mode strict

fix frozen_val = freeze(42)
// flux reassigned = frozen_val
// phase error: cannot bind crystal value with flux
//              for 'reassigned'
```

In strict mode, the phase checker tracks the phase of every expression and ensures that bindings are consistent. Binding a crystal expression with `flux` is an error because it would create a contradiction: the value is crystal but the binding says it should be fluid.

11.9.5 Strict Mode: Freezing an Already Crystal Value

Listing 11.43: Double-freeze error in strict mode

```
#mode strict

fix data = freeze([1, 2, 3])
// fix again = freeze(data)
// phase error: strict mode: cannot freeze an already
//              crystal value
```

This error helps catch redundant operations. If a value is already crystal, freezing it again is pointless—and in strict mode, Lattice tells you so.

11.9.6 Strict Mode: Thawing an Already Fluid Value

Listing 11.44: Thawing a fluid value in strict mode

```
#mode strict

flux data = [1, 2, 3]
// flux copy = thaw(data)
// phase error: strict mode: cannot thaw an already
//              fluid value
```

Similarly, thawing a fluid value is redundant. If you want an independent copy of a fluid value, use `clone()` instead.

11.9.7 Strict Mode: Fluid Values in `spawn`

Listing 11.45: Fluid value crossing spawn boundary

```
#mode strict

flux shared_counter = 0

scope {
    spawn {
        // print(shared_counter)
        // phase error: strict mode: cannot use fluid binding
        //   'shared_counter' across thread boundary in spawn
    }
}
```

The phase checker in `src/phase_check.c` includes a special `pc_check_spawn_expr()` function that walks expressions inside `spawn` blocks and errors if any fluid binding is referenced. This is one of the most valuable strict mode checks: it prevents data races at compile time.

11.9.8 Annotation Violations

Listing 11.46: Phase annotation violation

```
#mode strict

flux live = [1, 2, 3]

@crystal
fix snapshot = live
// phase error: @crystal annotation violated: initializer
//              for 'snapshot' is fluid
```

Phase annotations (`@fluid`, `@crystal`) are assertions about what phase a value should have. The checker tests them against the *initializer expression*, not against the binding keyword—so `@crystal flux x = 72.5` passes without complaint (a float literal is unphased, and unphased contradicts nothing), while assigning a known-fluid binding under a `@crystal` annotation does not. We cover annotations in detail in Chapter 14.

Reading Phase Errors

Phase errors always tell you three things:

1. **What went wrong**: the specific phase violation.
2. **Where it happened**: the variable name or expression involved.
3. **Why it matters**: strict mode errors explain the rule being enforced.

When you see a phase error, ask yourself: "Should this value be mutable here, or should I have frozen it earlier?" The answer usually reveals the right fix.

11.10 Querying Phase at Runtime

The built-in `phase_of()` function returns the current phase of any value as a string:

Listing 11.47: Using `phase_of()` for runtime inspection

```
flux counter = 0
fix pi = 3.14159
let name = "Lattice"

print(phase_of(counter))  // "fluid"
print(phase_of(pi))       // "crystal"
print(phase_of(name))     // "unphased"
```

The four possible return values are:

- `"fluid"` — the value is mutable (`VTAG_FLUID`).
- `"crystal"` — the value is immutable (`VTAG_CRYSTAL`).
- `"unphased"` — the value has no explicit phase (`VTAG_UNPHASED`).
- `"sublimated"` — the value is sealed against mutation and channel sends (`VTAG_SUBLIMATED`).

The `phase_of()` function is implemented as a native built-in registered in `src/runtime.c`. It inspects the `phase` field of the `LatValue` struct directly, making it a zero-cost query.

`phase_of()` is particularly useful in library code that needs to handle values of any phase:

Listing 11.48: Phase-aware library function

```
fn safe_append(collection: any, item: any) {
    if phase_of(collection) == "crystal" {
        // Can't modify crystal, return a new thawed copy
        flux copy = thaw(collection)
        copy.push(item)
        return freeze(copy)
    }
    // Fluid: append to our own copy and hand it back
    collection.push(item)
    return collection
}

flux items = [1, 2, 3]
flux grown = safe_append(items, 4)
print(grown)  // [1, 2, 3, 4]
print(items)  // [1, 2, 3]  -- the caller's array is untouched

fix frozen = freeze([10, 20])
fix updated = safe_append(frozen, 30)
print(updated)  // [10, 20, 30]
```

That second `print` is the important one, and it has nothing to do with phases. Arguments arrive by value, so the `collection.push(item)` in the fluid branch grows the callee's copy, not the caller's array. Phase-aware code still has to return its result—which, once you are in the habit, is the shape you wanted anyway.

11.11 Summary: The Phase Lifecycle

Let us put it all together. A value's phase lifecycle in Lattice looks like this:

1. A value is **created** via a constructor, literal, or expression. It begins as `VTAG_UNPHASED`.
2. When bound with `flux`, it transitions to `VTAG_FLUID` and lives on the FluidHeap under GC management.
3. When bound with `fix` or passed through `freeze()`, it transitions to `VTAG_CRYSTAL` and may be relocated to a CrystalRegion.
4. `thaw()` creates a *new* fluid copy from a crystal value. `clone()` creates a copy at the same phase.

5. `sublimate()` converts to `VTAG_SUBLIMATED`, the strictest in-place lock. `thaw()` still hands back a fluid copy of one (Section 11.6.4).

6. `forge`, `anneal`, `crystallize`, and `borrow` provide scoped and structured phase transitions.

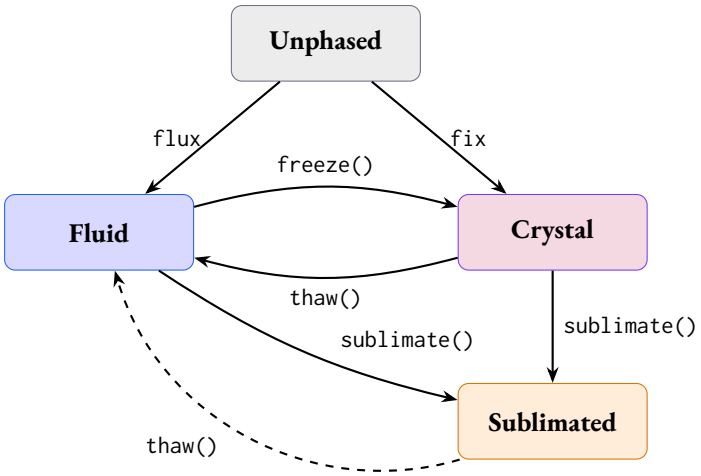


Figure 11.1: The phase transition diagram for Lattice values. The dashed edge is the one readers usually do not expect: `thaw()` melts a sublimated value too (Section 11.6.4).

11.12 Exercises

1. **Phase Explorer.** Write a function `describe_phase(val: any)` that takes any value and prints a message describing its phase. Include different messages for fluid, crystal, unphased, and sublimated values. Test it with values of each phase.

2. **Immutable Stack.** Implement a stack data structure using frozen arrays. Your `push` operation should return a *new* frozen array with the element added, and your `pop` operation should return a tuple of the top element and the remaining frozen array. The original stack should never change.

3. **Configuration Builder.** Write a `build_config()` function that uses a `forge` block to construct a configuration map. The function should accept an array of key-value tuples and build the frozen map from them.

4. **Phase-Safe Merge.** Write a function `merge_arrays(a: any, b: any)` that concatenates two arrays regardless of their phases. If either input is crystal, the function should thaw it before merging. The result should always be frozen.

5. **Error Collector.** In strict mode, write a program that deliberately triggers each of the phase errors described in Section 11.9 (one at a time). Comment out each error after confirming the message, and write a brief comment explaining what the error means.

What's Next. Now that we understand the phase system's semantics, we are ready to peer beneath the surface and ask: *where do these values actually live?* In Chapter 12, we explore the dual-heap memory architecture that makes the phase system possible—the FluidHeap for mutable values, the CrystalRegion for frozen data, and the ephemeral BumpArena for short-lived temporaries. Understanding this architecture will give you the intuition to write code that is not only correct but efficient.

Chapter 12

Memory and Arenas

In the previous chapter we learned *what* phases mean for your code. Now we ask: *where does the data actually go?* When you freeze a value, what happens to the bytes? When the garbage collector runs, how does it know to skip crystal values? Why does freezing sometimes make your program *faster*?

Lattice uses a *dual-heap architecture* that physically separates mutable and immutable data into different regions of memory. This is not a coincidence or an optimization afterthought—it is a direct consequence of the phase system's design. Fluid values live in one world, crystal values in another, and the boundary between them is enforced all the way down to the allocator.

The header comment at the top of `include/memory.h` states the invariant that makes the whole thing work: a value's backing store and the collector's reach are decided by *phase*, not by scope, lifetime, or where the variable was written. Freeze something and you have not merely re-labelled it—you have moved it out of the garbage collector's jurisdiction entirely.

12.1 The FluidHeap: Mark-Sweep GC for Mutable Values

Every mutable value in Lattice lives on the *FluidHeap*—a garbage-collected heap that uses a mark-sweep algorithm to reclaim memory.

When you write a `flux` declaration, the value's internal data (strings, array buffers, map entries) is allocated through the FluidHeap's `fluid_alloc()` function. You can see the implementation in `src/memory.c`:

Listing 12.1: Fluid values live on the GC-managed heap

```
flux names = ["Alice", "Bob", "Charlie"]
// The array buffer and each string are allocated via fluid_alloc()
// The FluidHeap tracks every allocation in a linked list
```

12.1.1 How the FluidHeap Works

The `FluidHeap` structure, defined in `include/memory.h`, maintains a linked list of every allocation it has made:

- Each allocation is wrapped in a `FluidAlloc` node containing the pointer, its size, a `marked` flag, and a link to the next allocation.
- The heap tracks `total_bytes`, `alloc_count`, and `gc_threshold` (initially 1 MB).
- When the total bytes exceed the GC threshold, a collection cycle is triggered.

The allocation path is straightforward: `fluid_alloc()` calls `malloc()`, wraps the result in a `FluidAlloc` header, prepends it to the linked list, and updates the byte counters. Deallocation via `fluid_dealloc()` walks the list, finds the matching pointer, unlinks the node, and frees the memory.

12.1.2 Mark-Sweep Collection

The GC uses a classic two-phase algorithm:

1. **Mark phase.** Starting from the *roots* (the VM stack, global environment, struct metadata, open upvalues, call frame upvalues, and the module cache), the collector recursively traverses every reachable value and marks its heap allocations. The `gc_mark_value()` function in `src/gc.c` handles this traversal.
2. **Sweep phase.** The collector walks the entire `FluidHeap` allocation list. Any allocation not marked in step 1 is unreachable—it gets freed. Marked allocations have their mark bit cleared for the next cycle.

Listing 12.2: GC reclaims unreachable fluid values

```
fn create_temporary() {
    flux temp = [1, 2, 3, 4, 5]
    // temp is allocated on the FluidHeap
    return temp.len()
    // After return, temp is unreachable
    // The next GC cycle will free its memory
}

flux length = create_temporary()
print(length)  // 5
```

Region-Backed Values Are Invisible to the GC

A critical optimization: `gc_mark_value()` in `src/gc.c` checks the `region_id` field of every value. If the region ID is anything other than `REGION_NONE`—meaning the value is region-backed, ephemeral, interned, or a constant—the marker returns immediately. Values living inside a crystal region are *never traversed* during the mark phase. This means the GC's pause time scales with the number of *fluid* values, not the total number of values. The more data you freeze, the less work the GC has to do.

The converse deserves saying too, because it is the part people get wrong. Not every frozen value moves into a region (Section 12.3). A crystal that stayed on the fluid heap keeps `region_id == REGION_NONE` and is marked and swept like any other allocation—its immutability rests on the phase tag alone. Freezing a five-character string buys you safety, not a smaller mark phase.

12.1.3 The GC Threshold and Adaptive Growth

The garbage collector does not run on every allocation. Instead, it triggers when the object count exceeds the `next_gc` threshold. After each collection, the threshold adapts:

$$\text{next_gc} = \text{surviving_objects} \times 2$$

This growth factor (defined as `GC_GROWTH_FACTOR` in `src/gc.c`) ensures that the GC runs more frequently when there are many objects and less frequently when most objects survive. The minimum threshold is 256 objects (`GC_INITIAL_THRESHOLD`), preventing thrashing on small programs.

Listing 12.3: Adaptive GC in action

```
flux items = []
for i in 0..10000 {
    items.push(to_string(i))
    // The GC may trigger periodically as items grows
    // Each cycle, the threshold adjusts based on survivors
}
print(items.len())  // 10000
```

12.2 The CrystalRegion: Arena-Backed Storage for Immutable Values

When you freeze a value—a large enough one, of the right shape—its data is copied into a *CrystalRegion*: an arena-based allocator that provides three benefits, cache locality, O(1) bulk deallocation, and, since v0.6.0, free aliasing.

12.2.1 What Is an Arena?

An arena (also called a region allocator or bump allocator) is a memory management strategy where you allocate objects by bumping a pointer forward within a pre-allocated block of memory called a *page*. When you are done with the arena, you free all its memory at once by freeing the pages. Individual objects are never freed independently.

Arena Allocation

An *arena* allocates memory by advancing a pointer within contiguous pages. Allocation is O(1)—just bump the pointer. Deallocation is also O(1)—free all pages at once. The trade-off: you cannot free individual objects; you free the entire arena or nothing.

This is a perfect fit for crystal values. Once frozen, they are immutable and their lifetimes are tied together: when the last reference to a frozen value is dropped, the entire region can be reclaimed.

12.2.2 Region Structure

The `CrystalRegion` struct, defined in `include/memory.h`, contains:

- An `id` (a `RegionId`) and an `epoch` timestamp.
- A linked list of `ArenaPage` structures, each backed by a 4 KB data buffer (`ARENA_PAGE_SIZE`).

- A running total of bytes used across all pages.
- An atomic reference count `rc`, plus `shared` and `sealed` flags.

Allocation within a region is handled by `arena_alloc()` in `src/memory.c`. It uses 8-byte alignment and tries to fit the request in the current head page. If the page is full, a new page is allocated and prepended to the list. Oversized requests get a dedicated page.

Listing 12.4: Freeze allocates into a CrystalRegion

```
flux large_dataset = []
for i in 0..1000 {
    large_dataset.push(i * i)
}

// freeze() copies the data into a new CrystalRegion
fix snapshot = freeze(large_dataset)

// The 1000 integers and the array buffer are now packed
// into contiguous arena pages---great for cache performance
print(snapshot.len())  // 1000
```

12.2.3 How Freeze Moves Data

`value_freeze_to_region()` in `src/value.c` is the entry point. It performs these steps:

1. **Eligibility.** `value_is_shareable()` walks the value rejecting closures, `Ref` cells, iterators, channels, and sublimated members. `value_worth_regionizing()` then rejects scalars and strings shorter than `REGION_SHARE_MIN_STR_LEN` (32 bytes). A rejection at either gate means a plain phase-tag flip and an early return—the value stays on the fluid heap.
2. **Region creation.** `crystal_region_create_shared()` mints a region with `rc = 1` and registers it in a process-global table.
3. **Arena activation.** The arena pointer `g_arena` (thread-local outside wasm builds) is set to the new region.
4. **Force-copy.** `value_clone_impl(v, false)` copies the value. Because `g_arena` is set, every internal allocation (`lat_alloc`, `lat_strdup`, ...) routes to `arena_alloc()`. The `false` says *do not share*: nested crystals are copied *into* this region rather than aliased, so no region ever points at another one. Refcounting stays trivially cycle-free.
5. **Arena deactivation.** `g_arena` is restored.

6. **Tagging.** `region_tag_recursive()` stamps `VTAG_CRYSTAL` and the region handle onto every node in the copy, and normalizes away any stale per-key or per-field phase metadata.

7. **Sealing.** `crystal_region_seal()` `mprotect`s the pages read-only. This is a debug backstop—a no-op in release builds and on platforms without it—so that a stray write during development dies loudly instead of quietly.

8. **Release the original.** The pre-freeze value is freed, releasing any nested shared handles it held, and the region-backed copy takes its place.

The allocation routing in `src/value.c` is what makes step 4 transparent:

- `lat_alloc()` checks `g_arena` first. If set, it calls `arena_alloc()`. Otherwise, it tries the FluidHeap, and as a last resort, raw `malloc()`.

- `lat_free()` checks `g_arena`—if set, it is a no-op (arena memory is freed in bulk).

The Region ID Is a Pointer

A region-backed value does not store a region *number*. It stores the `CrystalRegion*` itself with the low bit set—see `REGION_TAG` and `REGION_IS_SHARED_ID` in `include/value.h`. The trick is sound because regions are `malloc`'d and therefore at least 8-byte aligned, so the low bit is always free; the odd sentinels `REGION_NONE` and `REGION_INTERNED` are excluded by name, and the even ones fail the bit test outright. Because it tests alignment rather than an address range, the same predicate holds on wasm32 and Windows. Following a value to its region is a mask, not a lookup.

12.2.4 Region Lifetime: Refcounts, Not Collection

Here is where the architecture parts company with most garbage-collected runtimes. Crystal regions are **not** collected. They are reference counted, and the collector never touches them.

Every alias of a region-backed value—an assignment, an argument pass, a return, a channel send, a spawn capture—is a bitwise copy of the handle plus `crystal_region_retain()`, an atomic increment. Every `value_free()` of such a handle is `crystal_region_release()`, an atomic decrement. When the count reaches zero, the region's page list is walked once and freed. That is the entire lifecycle.

Listing 12.5: Aliasing a region is a refcount bump

```
fn make_snapshot() {
    flux data = []
    for i in 0..5000 { data.push(i) }
    return freeze(data)
}

fix snapshot = make_snapshot()
print(snapshot.len())  // 5000

// Each of these is O(1)---one atomic increment, no copy
fix a = snapshot
fix b = snapshot
print(a.len() + b.len())  // 10000
```

The pages themselves come from plain global `malloc`, deliberately never routed through any thread's FluidHeap. That is what lets a region outlive the thread that created it: a worker can freeze a result, send it to its parent, and exit, and the last release may safely run on a completely different thread.

Why the GC Cannot Help Here

The collector has no view of a region's true reachability. A live handle can sit in a channel buffer, a history snapshot, the module cache, or a sibling thread's evaluator—places GC roots do not reach. So the GC is deliberately given no authority: in debug builds it can *report* regions it cannot see (`crystal_region_report_unreachable()`), advisory only, and it never frees one. Refcounting is the sole reclaimer. The O(1) page free is sound precisely because regions contain pure data—the shareability scan in Section 12.2.3 guarantees there are no channels, refs, or iterators inside needing finalization.

12.2.5 The RegionManager

The `RegionManager` in `include/memory.h` is the older bookkeeper: a dynamic array of regions, a `next_id` counter, an epoch counter, and statistics (total allocations, peak region count, cumulative data bytes). Its `region_collect()` walks a sorted set of reachable region IDs with a binary search and frees the rest.

It is worth knowing about because it is still part of the `DualHeap` structure and still exercised by the unit tests—but as of v0.6.0 the evaluator no longer creates regions through it. Shared, refcounted regions superseded it. If you go reading `src/memory.c` and find two apparently competing region lifecycles, that is why: one is the current mechanism, the other is the scaffolding it grew out of.

12.3 Two Kinds of Crystal

Not everything you freeze ends up in a region. The runtime keeps two internal representations of a crystal value, and which one you get is decided at freeze time by the two gates from Section 12.2.3.

Shared vs. Legacy Crystal

A *shared crystal* is a `VTAG_CRYSTAL` value whose backing store lives entirely inside one sealed, refcounted region; its `region_id` is a tagged `CrystalRegion*`. Aliasing it is a bitwise handle copy plus an atomic retain.
A *legacy crystal* is a `VTAG_CRYSTAL` value with `region_id == REGION_NONE`. It lives on the fluid heap, is marked and swept like any other allocation, and copies by full deep clone. Its immutability rests on the phase tag alone.

A value takes the legacy path when it is unshareable or not worth the trouble:

- **Unshareable** (`value_is_shareable()`): the value's graph contains a closure, a `Ref`, an iterator, a channel, or a sublimated member anywhere inside it. These carry non-atomic foreign refcounts or thread-confined state, and admitting them would break both the purity of the region and the soundness of the O(1) page free.
- **Not worth regionizing** (`value_worth_regionizing()`): scalars—`Int`, `Float`, `Bool`, `Unit`, `Nil`, ranges—and strings shorter than 32 bytes. Copying these outright is cheaper than the refcount traffic would be.
- **Partial freezes**: a map frozen with `except` keeps a writable per-key override table, and sealed pages are not writable. It stays legacy on purpose.

Everything else—arrays, maps, structs, sets, tuples, buffers, enums, and strings of 32 bytes or more—materializes into a region.

You Cannot Tell Which You Have

There is no way to observe the difference from Lattice code, and that is a load-bearing property, not an accident. Sharing is only safe because v0.5.0 made frozen values genuinely unmutable on every backend: a value that provably cannot change is a value whose aliasing nobody can detect. The repository ships a differential mode, `LATTICE_FORCE_COPY=1`, that turns every would-be share back into a deep copy; program output must be bit-identical with it on and off.

12.3.1 Turning Sharing Off

Three switches control the mechanism, and all three exist for debugging rather than tuning:

- `clat -no-regions` — disables region materialization wholesale. Every freeze becomes a plain tag flip. Spawned threads inherit the setting.
- `LATTICE_SHARE_CRYSTALS=0` — the same effect as a live environment variable, for embedded hosts and the `clat-run` thin runtime, where there is no command line to pass a flag on. It is read fresh at each freeze (`share_disabled_by_env()` in `src/value.c`), so it can be flipped mid-process; handles already shared stay valid and refcounted, and only *new* freezes stop materializing. Only the exact string `0` disables it.
- `LATTICE_FORCE_COPY=1` — the differential oracle above.

One Live Element Pins the Whole Region

A region is freed as a unit. Pull a single element out of a large frozen structure and hold onto it, and the handle you are holding keeps every page of that region alive—the other 9,999 elements included. `clone(elem)` takes a physically independent copy and cuts the tie. This is the one place where the sharing model can surprise you in the direction of *more* memory, and it is worth remembering when you are extracting a small field from a big frozen record and stashing it somewhere long-lived.

12.3.2 Crystallized Maps

Regionizing is not the only thing freeze does to a large map. When a map is fully frozen, carries no per-key phase overrides, and holds at least `CMI_MIN_KEYS` (16) live keys, `value_crystallize()` in `src/value.c` hands it to `lat_map_crystallize()` in `src/ds/hashmap.c`, which rebuilds the sparse open-addressing table into a single contiguous block: a packed entry array, a displacement table, and the key bytes, laid out as a CHD minimal perfect hash with a bucket load factor of four.

Lookups in that layout are one displacement load and one packed load. Iteration order is preserved exactly—the build walks the occupied slots in scan order and keeps them in that order—so nothing observable changes.

`thaw()` undoes it: `value_decrystallize()` rebuilds the sparse table, because a mutable map needs somewhere to put new keys. Adding a key to a crystallized map degrades it back to the sparse layout for the same reason.

"Freeze Is Just a Tag Change" Is Two Versions Out of Date

Between region materialization and map crystallization, freezing a large structure now does real work: it copies, it seals, and it may rebuild a hash table. The payoff is on the other side—aliasing becomes free, and iteration over a big frozen map gets substantially faster. Freeze once, at a boundary, and let everything downstream be cheap.

12.4 The Ephemeral BumpArena: Short-Lived Temporaries

Not every allocation deserves the overhead of GC tracking or the permanence of an arena region. For short-lived temporary values—intermediate computation results, format strings, temporary buffers—Lattice provides the `BumpArena`.

The BumpArena is a lightweight bump allocator that can be *reset* without freeing its pages. This means the same memory pages are reused across multiple reset cycles, avoiding the cost of repeated page allocation.

12.4.1 How the BumpArena Works

The `BumpArena` struct in `include/memory.h` maintains:

- A `pages` pointer to the current page.
- A `first_page` pointer to the head of the page chain (preserved across resets).
- A running total of bytes allocated since the last reset.

Allocation via `bump_alloc()` (in `src/memory.c`) works in three steps:

1. Try the current page. If there is room, bump the `used` pointer and return.
2. Try the next page in the chain (left over from a previous cycle). If there is room, advance to that page.
3. Allocate a new page. Oversized requests get a dedicated page larger than the default 4 KB.

All allocations are 8-byte aligned via `(size + 7) & ~7`, ensuring proper alignment for any data type.

12.4.2 The Reset Cycle

The key feature of the BumpArena is `bump_arena_reset()`. Instead of freeing pages, it simply sets each page's `used` counter back to zero and resets the current page pointer to the first page. The page chain remains intact, so subsequent allocations reuse existing memory without any system calls.

Listing 12.6: Ephemeral allocations for temporaries

```
// Inside the VM, each expression evaluation may allocate
// temporary strings for interpolation, display, or hashing.
// These temporaries live in the BumpArena and are cleared
// at safe points (e.g., between statements).

flux message = "Count: ${to_string(counter)}"
// The interpolation creates temporary strings in the BumpArena
// They are freed in bulk at the next safe point
```

Values allocated in the BumpArena have their `region_id` set to `REGION_EPHEMERAL` (defined as `(size_t)-2` in `include/value.h`). This sentinel value tells both the GC marker and the value destructor to skip these allocations—they are managed by the arena lifecycle, not by normal heap operations.

Performance Benefit

The BumpArena is one of the reasons Lattice can handle string-heavy operations efficiently. String interpolation, formatting, and display operations generate many short-lived strings. Rather than allocating and freeing each one individually (which would pressure the GC), they are bump-allocated and cleared in bulk. This reduces GC pressure and improves throughput.

12.5 How Freeze Moves a Value from Heap to Arena

We took the machinery apart in Section 12.2.3. Now let us watch it run on one concrete value, because the interesting part is what is left behind.

12.5.1 Step by Step

Consider this code:

Listing 12.7: A freeze in slow motion

```
flux temperatures = [20.1, 21.5, 19.8]
fix snapshot = freeze(clone(temperatures))

print(phase_of(temperatures))  // "fluid"
print(phase_of(snapshot))      // "crystal"
```

Here is the memory journey:

1. **Initial state.** `temperatures` is a fluid array. Its element buffer (`elems`) is allocated on the FluidHeap. The three float values are stored inline in the buffer. The `phase` tag is `VTAG_FLUID`, and `region_id` is `REGION_NONE`.

2. **The clone.** `clone()` produces a second fluid array on the FluidHeap—a full copy, and the reason `temperatures` survives this listing as a fluid binding at all.

3. **Eligibility.** The clone is an array of floats: no closures, no refs, nothing thread-confined, and not a scalar. Both gates pass, so it will materialize.

4. **Region creation.** `crystal_region_create_shared()` mints a sealed region with `rc = 1` and a single 4 KB page.

5. **Arena activation.** The `g_arena` pointer is set to the new region. From this point, all `lat_alloc()` calls are routed to `arena_alloc()`.

6. **Force-copy.** The array is copied again, this time into the arena. The new element buffer lands in the region's page, not the FluidHeap. Compound elements recurse; nested crystals are copied in rather than aliased.

7. **Arena deactivation.** `g_arena` is restored. Subsequent allocations go through the normal FluidHeap path.

8. **Tag and seal.** `region_tag_recursive()` stamps `VTAG_CRYSTAL` and the tagged region handle onto every node, and the pages are sealed read-only in debug builds.

9. **Release.** The pre-freeze clone is freed. Its FluidHeap buffer is gone; only the region copy remains.

10. **Binding.** The region-backed handle is stored in `snapshot`.

Two copies for one freeze, then—one from `clone()`, one into the arena. That is the shape of the cost, and it is why Section 11.8 quotes materialization at roughly 1.9× a bare tag flip. It is also why you pay it once, at a boundary, and never again: from step 10 onward, every alias of `snapshot` is a single atomic increment.

Drop the `clone()` and step 2 disappears along with `temperatures`' fluid phase—the freeze writes back into the binding, as Section 11.5.1 explains, and you are left with one array in one region.

Property	FluidHeap	CrystalRegion
Allocation style	Individual `malloc()`	Arena bump allocation
Deallocation	Per-object via GC sweep	Bulk page free at `rc == 0`
Cache locality	Scattered	Contiguous
GC interaction	Mark & sweep	Skipped during marking
Copy cost	Deep clone	O(1) handle + atomic retain
Mutability	Mutable	Immutable (pages sealed in debug builds)
`region_id`	`REGION_NONE`	Tagged `CrystalRegion*`

Table 12.1: Comparing FluidHeap and CrystalRegion memory characteristics.

12.5.2 Memory Layout Comparison

Legacy Crystals Sit in the Left Column

The right-hand column describes a *shared* crystal. A legacy crystal—an unshareable graph, a short string, a partially frozen map—has every property in the left column except the last row of Table 12.1: it is immutable, but it is FluidHeap-resident, GC-traced, and deep-cloned on copy.

12.6 String Interning and Why It Matters

Lattice interns certain strings—struct field names, map keys used internally, and explicitly interned strings—into a global `InternTable`. Interning means that each unique string exists at most once in memory, and all references to that string point to the same canonical copy.

12.6.1 How Interning Works

The intern table, implemented in `src/intern.c`, is a hash table using open addressing with linear probing:

- The hash function is DJB2 (`h = h * 33 + c`), a fast string hash.
- The table starts at 256 entries and grows (doubles) when the load factor exceeds 50%.
- `intern()` looks up the string. If found, it returns the existing pointer. If not, it `strdup()`s the string into the table and returns the new pointer.

The result: two interned strings with the same content are guaranteed to have the same pointer. This means pointer comparison (==) replaces `strcmp()` for interned strings, turning an O(n) comparison into an O(1) operation.

12.6.2 What Gets Interned

In the Lattice runtime, the following strings are automatically interned:

- **Struct field names.** When a struct is created (via `value_struct()` in `src/value.c`), each field name is passed through `intern()`. This is visible in the constructor: `val.as.strct.field_names[i] = (char *)intern(field_names[i])`.
- **Explicitly interned strings.** The `value_string_interned()` constructor creates a string value whose `region_id` is set to `REGION_INTERNED`.

Interned strings are never freed during normal program execution (they are owned by the global intern table) and are cleaned up only when `intern_free()` is called at program exit.

12.6.3 Why Interning Matters for Phase

String interning interacts with the phase system in several important ways:

1. **GC safety.** Interned strings have `region_id = REGION_INTERNED`. The GC marker skips them (they are not on the FluidHeap), and the value destructor skips them (they are owned by the intern table). This prevents double-frees and dangling pointers.
2. **Efficient deep cloning.** When `value_deep_clone()` encounters an interned string, it does *not* copy the string data. Instead, it reuses the same pointer and copies the `REGION_INTERNED` marker. This makes cloning structs (which have interned field names) cheaper.
3. **Fast field lookup.** Because field names are interned, struct field access can use pointer comparison instead of string comparison, speeding up field lookups in hot paths.

Listing 12.8: Interning makes struct operations faster

```
struct Point {
    x: Int,
    y: Int
}

// The field names "x" and "y" are interned once.
// Every Point instance shares the same name pointers.
// Field lookup compares pointers, not strings.

flux p1 = Point { x: 10, y: 20 }
flux p2 = Point { x: 30, y: 40 }
// p1.as.strct.field_names[0] == p2.as.strct.field_names[0]
// (same pointer, thanks to interning)
```

12.7 The DualHeap: Putting It All Together

The `DualHeap` struct in `include/memory.h` ties the FluidHeap and RegionManager together:

- `fluid` — a pointer to the `FluidHeap` for GC-managed mutable allocations.
- `regions` — a pointer to the `RegionManager` for arena-backed crystal storage.

At VM startup, `dual_heap_new()` creates both subsystems. At shutdown, `dual_heap_free()` tears them down, freeing all remaining fluid allocations and all crystal regions.

The DualHeap is set as the active heap via `value_set_heap()` at the start of program execution. This call installs the `DualHeap` into a thread-local global, so all subsequent `lat_alloc()` calls are routed through it.

Listing 12.9: The DualHeap manages both worlds

```
// At program startup:
// DualHeap is created
//   -> FluidHeap for flux values
//   -> RegionManager for fix values

flux mutable_data = [1, 2, 3]     // allocated via FluidHeap
fix frozen_data = freeze([4, 5])  // cloned into CrystalRegion

// At program shutdown:
// DualHeap frees everything
//   -> FluidHeap frees all remaining fluid allocations
//   -> RegionManager frees all crystal regions and their pages
```

12.8 GC Flags: Tuning the Collector

Lattice provides command-line flags to control garbage collection behavior. These are useful for debugging memory issues, benchmarking, and tuning performance.

12.8.1 –gc: Enable the Garbage Collector

By default, the GC is enabled in the bytecode VM. The `--gc` flag explicitly enables garbage collection:

Listing 12.10: Running with GC enabled

```
$ clat --gc my_program.lat
```

The GC state is managed by the `GC` struct in `include/gc.h`, which tracks:

- `all_objects`: linked list of all GC-tracked allocations.
- `object_count`: current number of tracked objects.
- `next_gc`: threshold for the next collection.
- `enabled`: whether the GC is active.
- `total_collected` and `total_cycles`: lifetime statistics.

12.8.2 `--gc-stress`: Collect on Every Allocation

The `--gc-stress` flag enables *stress mode*, where the collector runs on every single allocation. This is invaluable for catching GC-related bugs: use-after-free, missing root registrations, and incorrect mark implementations.

Listing 12.11: Stress testing the GC

```
$ clat --gc-stress my_program.lat

# Every allocation triggers a full mark-sweep cycle.
# Extremely slow, but catches subtle memory bugs.
```

In stress mode, `gc_maybe_collect()` in `src/gc.c` always triggers a collection:

```
if (gc->stress || gc->object_count >= gc->next_gc) { gc_collect(gc,
vm_ptr); }
```

Do Not Use Stress Mode in Production

GC stress mode runs a full mark-sweep cycle on every allocation, making it orders of magnitude slower than normal execution. Use it only for debugging and testing.

12.8.3 `--gc-incremental`: Spread Work Across Safe Points

The `--gc-incremental` flag enables *incremental collection*, which spreads GC work across multiple safe points to reduce pause times.

Instead of stopping the world for a full mark-sweep, the incremental collector operates as a state machine with four phases, defined as `GCPhase` in `include/gc.h`:

1. **Idle** (`GC_PHASE_IDLE`). No collection in progress. The collector checks if the threshold is exceeded and, if so, transitions to marking.

2. **Mark roots** (`GC_PHASE_MARK_ROOTS`). Scans all VM roots (stack, globals, upvalues, module cache) and pushes compound values onto a *gray worklist*. Leaf types (strings, buffers) are marked directly without being pushed.

3. **Mark trace** (`GC_PHASE_MARK_TRACE`). Pops values from the gray worklist and traces their children, up to a configurable `mark_budget` per step (default: 64 values). When the worklist is empty, the roots are re-scanned once (to catch mutations during marking), then the sweep begins.

4. **Sweep** (`GC_PHASE_SWEEP`). Walks the object list, freeing unmarked objects, up to a `sweep_budget` per step (default: 128 objects). When the sweep completes, the cycle is done and the threshold is updated.

Listing 12.12: Incremental GC for lower latency

```
$ clat --gc-incremental my_program.lat

# The GC spreads its work across VM safe points
# (typically at OP_RESET_EPHEMERAL boundaries),
# keeping individual pauses short.
```

No Write Barriers

Lattice's incremental collector does not use write barriers. Instead, it compensates by re-scanning roots after the initial mark trace completes (the `roots_rescanned` flag in the `GC` struct). This conservative approach is correct but may do slightly more work than a write-barrier-based collector. New objects allocated during an incremental cycle are born "black" (marked), so the sweep phase will not prematurely free them.

12.8.4 GC Statistics

The `GC` struct tracks several statistics that are useful for performance analysis:

Similarly, the RegionManager tracks:

12.9 Special Region IDs

Every `LatValue` has a `region_id` field that indicates where its data lives. The runtime defines several sentinel values (in `include/value.h`) that identify special allocation regions:

Field	Meaning
`object_count`	Current number of GC-tracked objects
`bytes_allocated`	Total bytes currently allocated under GC
`total_collected`	Lifetime count of objects freed by the GC
`total_cycles`	Number of GC cycles completed
`next_gc`	Object count threshold for next collection

Table 12.2: GC statistics fields.

Field	Meaning
`count`	Number of live crystal regions
`total_allocs`	Lifetime count of regions created
`peak_count`	Maximum simultaneous live regions
`cumulative_data_bytes`	Total bytes allocated across all regions

Table 12.3: RegionManager statistics fields.

Any `region_id` that is not one of these sentinels is a real CrystalRegion ID. The GC marker, the value destructor, and the deep-clone function all check `region_id` to decide how to handle a value.

12.10 Memory Best Practices

Armed with knowledge of the dual-heap architecture, here are practical guidelines for memory-efficient Lattice programs:

Constant	Value	Meaning
`REGION_NONE`	`(size_t)-1`	Normal heap (FluidHeap or `malloc`)
`REGION_EPHEMERAL`	`(size_t)-2`	BumpArena (temporary, reset-managed)
`REGION_INTERNED`	`(size_t)-3`	Intern table (never freed individually)
`REGION_CONST`	`(size_t)-4`	Constant pool (borrowed, not freed)

Table 12.4: Special region ID sentinel values.

Memory Guidelines

- **Freeze early, freeze often.** Crystal values reduce GC pressure because the marker skips them entirely. If data will not change after initialization, freeze it.
- **Use forge blocks for complex construction.** Building a value in a forge block localizes the mutable temporaries and produces a single crystal result in one arena, maximizing cache locality.
- **Avoid unnecessary thaw-modify-freeze cycles.** Each cycle involves a deep clone. If you need to make multiple modifications, thaw once, apply all changes, and freeze once.
- **Let large fluid values die quickly.** The GC threshold adapts based on surviving objects. If large temporary fluid values are kept alive unnecessarily, the GC threshold grows and memory usage stays high. Drop references as soon as you are done.
- **Use incremental GC for latency-sensitive applications.** If your program handles real-time events (web servers, game loops), `--gc-incremental` keeps pauses bounded.

12.11 Exercises

1. **Freeze Cost.** Write a program that creates a deeply nested array (arrays of arrays of arrays, at least 3 levels deep with 100 elements at each level). Time how long it takes to freeze the entire structure. Then time how long it takes to access elements of the frozen copy versus the original fluid copy. What do you observe?

2. **Snapshot Chain.** Write a function that maintains a "history" of an evolving value by freezing a snapshot after every modification. Store the snapshots in an array. How many distinct CrystalRegions do you think are created?

3. **GC Pressure Experiment.** Write a program that creates many short-lived fluid values in a loop (e.g., build and discard 10,000 arrays of 100 elements each). Run it with `--gc` and observe the behavior. Then modify the program to freeze values that persist. Reason about how the GC threshold changes in each case.

4. **Intern Detective.** Create two structs of the same type with different field values. Use `phase_of()` and `typeof()` to inspect them. Reason about which parts of the structs share memory (interned field names) and which have independent allocations (field values).

What's Next. We have seen how individual values transition between fluid and crystal, and where they live in memory. But what about data structures where *some fields* should be mutable and *others* should be frozen? In Chapter 13, we explore alloy structs—structures with per-field phase declarations—and the reactive bond system that lets values respond automatically when their neighbors change phase.

Chapter 13

Alloys and Reactive Bonds

So far, the phase system has been all-or-nothing: a value is either entirely fluid or entirely crystal. Freeze an array, and every element freezes. Freeze a struct, and every field hardens. But real-world data is rarely so uniform.

Consider a user session object. The user's ID and email should be immutable once established—nobody should accidentally overwrite them. But the session's last-active timestamp needs to update on every request, and the permissions cache might need to refresh periodically. You need a struct where *some fields are crystal and others are fluid*—a material with mixed properties.

In materials science, such a material is called an *alloy*: a combination of elements with different characteristics fused into a single structure. Lattice borrows this metaphor. An alloy struct has per-field phase declarations, so each field can have its own mutability. And once you have mixed-phase data, Lattice goes further with *reactive bonds*—automatic relationships between variables that propagate phase changes—and *seeds*—validation contracts that guard the crystallization process.

13.1 Alloy Structs: Per-Field Phase Declarations

An alloy struct declares the phase of each field individually by writing `flux` or `fix` as part of the field's *type*, right after the colon:

Listing 13.1: Defining an alloy struct

```
struct Session {
    user_id: fix String,
    email: fix String,
    last_active: flux String,
    permissions: flux Array
}
```

Note the placement: the phase qualifier belongs to the type expression (`fix String`), not to the field name. Writing `fix user_id: String` is a parse error—the parser expects a field name first, then a colon, then a (possibly phase-qualified) type.

When you create an instance of this struct, the runtime applies per-field phase enforcement. The `fix` fields are automatically frozen, while the `flux` fields remain fluid:

Listing 13.2: Creating an alloy struct instance

```
flux session = Session {
    user_id: "usr_42",
    email: "alice@example.com",
    last_active: "2024-01-15T10:30:00Z",
    permissions: ["read", "write"]
}

// Crystal fields are locked
print(session.user_id)     // "usr_42"
// session.user_id = "usr_99"  // Error: cannot assign to frozen field 'user_id'

// Fluid fields can change
session.last_active = "2024-01-15T10:31:00Z"
print(session.last_active)  // "2024-01-15T10:31:00Z"
```

Attempting to write a crystal field is a runtime error on every backend:

Listing 13.3: Crystal fields reject assignment

```
session.user_id = "usr_99"
// vm error: cannot assign to frozen field 'user_id'
```

The core message is identical everywhere; only the prefix changes (`vm error:` on the default stack VM, `error:` under `--tree-walk`, `regvm error:` under `--regvm`). We will quote the core message from here on.

Nested Writes Through Struct Fields

As of v0.6.4, only *whole-field assignment* (`session.last_active = ...`) reliably writes back into a struct on every backend. Deeper writes through a field—`session.permissions.push("admin")` or `session.permissions[0] = "x"`—are silently discarded by the two bytecode backends (they mutate a copy of the field), while the tree-walker applies them. Until this divergence is resolved, use the read-modify-write idiom:

```
flux perms = session.permissions
perms.push("admin")
session.permissions = perms
```

This works identically on all three backends.
Note the asymmetry: v0.6.4 taught the bytecode VMs to write back through *array and map* elements, so `obj[0].push(9)` and `m["k"].push(2)` now mutate for real (Chapter 7). Struct *fields* did not get the same treatment.

Alloy

An *alloy* is a struct (or map) with per-field phase declarations. Individual fields carry their own phase tag, independent of the overall struct's phase. Crystal fields reject mutation; fluid fields allow it. The struct itself can be fluid (allowing field reassignment for fluid fields) while its crystal fields remain permanently locked.

13.1.1 How Alloys Work Under the Hood

The per-field phase system is implemented through the `field_phases` array in the `LatValue` struct (see `include/value.h`). When a struct has per-field phases, `field_phases` is a dynamically allocated array of `PhaseTag` values—one for each field. If `field_phases` is `NULL`, all fields inherit the struct's overall phase.

During struct construction in the VM (`src/stackvm.c`), the runtime checks for a per-field phase declaration stored in the global environment under the key `__struct_phases_<StructName>`. This is an array of integer phase codes:

- `0` means `PHASE_FLUID` — the field stays mutable.
- `1` means `PHASE_CRYSTAL` — the field is automatically frozen at construction.

- Any other value means the field inherits the struct's overall phase.

Those numbers are the `AstPhase` enumeration from `include/phase.h`, whose members are declared in the order `PHASE_FLUID`, `PHASE_CRYSTAL`, `PHASE_UNSPECIFIED`. Because struct literals list their fields in whatever order the author wrote them while the phase metadata is stored in declaration order, the VM matches the two by field *name* rather than position—so you can construct an alloy with its fields in any order you like.

When field assignment is attempted (`OP_SET_FIELD`), the VM checks the `field_phases` array:

Listing 13.4: Field-level phase enforcement

```
// Internally, the VM checks:
// if (obj.as.strct.field_phases[i] == VTAG_CRYSTAL)
//     ERROR: "cannot assign to frozen field 'user_id'"
```

This enforcement happens at the deepest level of the runtime—there is no way to bypass it through clever coding. The field's phase tag is a property of the value itself, checked on every write.

13.1.2 Partial Freeze with `except`

You can also create alloy-like behavior dynamically by freezing a struct with exceptions. The `except` clause follows the `freeze()` call:

Listing 13.5: Freezing a struct with exceptions

```
struct Config {
    host: String,
    port: Int,
    debug: Bool,
    log_level: String
}

flux config = Config {
    host: "localhost",
    port: 8080,
    debug: true,
    log_level: "info"
}

// Freeze everything except debug and log_level
freeze(config) except ["debug", "log_level"]

// config.host = "evil.com"   // Error: cannot assign to frozen field 'host'
config.debug = false           // OK: debug was exempted
config.log_level = "debug"     // OK: log_level was exempted
```

The `except` clause is compiled to the `OP_FREEZE_EXCEPT` opcode. Under the hood (in `src/stackvm.c`), this opcode:

1. Allocates the `field_phases` array if it does not already exist.

2. Iterates through each field, freezing it unless it appears in the exception list.

3. Sets the exempted fields' phases to `VTAG_FLUID`.

The same mechanism works for maps—each key can have its own phase via the `key_phases` map stored alongside the map's data. Exempt keys accept both index assignment and mutating methods, while every other key stays locked:

Listing 13.6: Freeze-except on a map (v0.6.4 semantics)

```
flux m = Map::new()
m.set("host", "localhost")
m.set("retries", 3)

freeze(m) except ["retries"]

m["retries"] = 5        // OK: exempt key
m.set("retries", 7)     // OK: set/remove on exempt keys allowed as of v0.6.4
// m["host"] = "y"      // Error: cannot modify a frozen value
```

Three rules complete the picture (all verified identical on the three backends):

- A key missing from the `except` list inherits the frozen map's phase—writes to it error.
- Whole-map mutators such as `merge` still error: the exemption is per-key, not per-map. The message names the offending method and offers the remedy—`cannot call mutating method 'merge' on crystal value 'm' (use thaw(m) to make it mutable).`
- A later full `freeze(m)` *closes the holes*: it drops all per-key overrides, and previously exempt keys become frozen like everything else. `thaw()` likewise discards per-key overrides.

Earlier releases (v0.5.0) blanket-blocked method calls like `set` even on exempt keys; v0.6.4 relaxed this so both write forms behave consistently.

13.2 Designing Data with Mixed Mutability

Alloy structs are a powerful design tool. Here are patterns that emerge naturally from per-field phases.

13.2.1 Identity + State Pattern

Separate immutable identity from mutable state:

Listing 13.7: Identity + State pattern

```
struct Player {
    id: fix Int,
    name: fix String,
    health: flux Int,
    position_x: flux Float,
    position_y: flux Float,
    inventory: flux Array
}

flux hero = Player {
    id: 1,
    name: "Aria",
    health: 100,
    position_x: 0.0,
    position_y: 0.0,
    inventory: ["sword", "shield"]
}

// Identity is locked forever
print(hero.name)  // "Aria"

// State changes freely
hero.health = hero.health - 15
hero.position_x = 3.5
flux inv = hero.inventory
inv.push("potion")
hero.inventory = inv
print(hero.inventory)  // [sword, shield, potion]
```

13.2.2 Configuration + Runtime Pattern

Lock configuration values while allowing runtime counters and caches to change:

Listing 13.8: Configuration + Runtime pattern

```
struct Server {
    host: fix String,
    port: fix Int,
    max_connections: fix Int,
    active_connections: flux Int,
    request_count: flux Int
}

flux server = Server {
    host: "0.0.0.0",
    port: 443,
    max_connections: 10000,
    active_connections: 0,
    request_count: 0
}

// Configuration is safe from accidental changes
// Runtime counters update freely
server.active_connections = server.active_connections + 1
server.request_count = server.request_count + 1
```

13.2.3 Append-Only Pattern

Use a crystal array field for historical records (by replacing it with a new frozen array), while keeping a mutable accumulator:

Listing 13.9: Append-only log pattern

```
struct AuditLog {
    entries: fix Array,
    pending: flux Array
}

fn add_entry(log: any, entry: any) {
    flux pending = log.pending
    pending.push(entry)
    log.pending = pending
    return log
}

fn flush_log(log: any) {
    flux all = thaw(log.entries)
    for item in log.pending {
        all.push(item)
    }
    // Return new log with frozen entries and empty pending
    return AuditLog {
        entries: freeze(all),
        pending: []
    }
}

flux log = AuditLog { entries: freeze(["created"]), pending: [] }
log = add_entry(log, "login")
log = flush_log(log)
print(log.entries)  // [created, login]
```

Structs are pass-by-value, so `add_entry` and `flush_log` return the updated log and the caller rebinds it—mutations inside the callee never propagate back on their own.

Alloy Design Principle

When designing an alloy struct, ask: "Which fields define *what this thing is* (identity), and which fields describe *how it is right now* (state)?" Identity fields should be `fix`. State fields should be `flux`.

13.3 Reactive Bonds: `bond()`, `unbond()`, `react()`, `unreact()`

Lattice provides a reactive system that lets variables respond automatically when the phase of another variable changes. The core primitives are:

- `react(variable, callback)` — register a callback that fires when a variable's phase changes.
- `unreact(variable)` — remove all reaction callbacks for a variable.
- `bond(source, dependent, strategy)` — link variables so that freezing `source` triggers a phase change in `dependent`.
- `unbond(source, dependent)` — remove a specific link.

All four take *bare variable names*, not strings—the compiler captures the identifier at the call site and never evaluates it. Quoting one is a runtime error, and which error depends on *which* argument you quoted. Quote the dependent—`bond(primary, "replica", "mirror")`—and the bytecode backends report `bond() requires variable names for dependencies` while `--tree-walk` words it `bond() requires variable names for first two arguments`. Quote the source instead—`bond("primary", replica, "mirror")`—and the two VMs never recognise the call as a bond at all, so it degrades into an ordinary function lookup and fails with `undefined variable 'bond' (did you mean 'round'?)`; only the tree-walker still names the real problem. The distinction is worth internalizing now, because the pressure and seed functions later in this chapter do not all follow the same convention. The one exception is the trailing `strategy` argument of `bond()`, which *is* an ordinary string literal.

13.3.1 Reactions: Responding to Phase Changes

`react()` registers a callback closure that is invoked whenever a variable undergoes a phase transition (freeze, thaw, or sublimate). The callback receives two arguments: the new phase as a string, and a snapshot of the variable's current value:

Listing 13.10: Setting up a phase reaction

```
flux temperature = 72.5

react(temperature, |phase, val| {
    print("Temperature phase changed to: " + phase)
})

freeze(temperature)
// Output: Temperature phase changed to: crystal

thaw(temperature)
// Output: Temperature phase changed to: fluid
```

The first callback argument is a string describing the new phase: `"crystal"`, `"fluid"`, or `"sublimated"`. The second is a deep clone of the value at the moment of transition—inspect it freely; mutating it affects nothing. Multiple callbacks can be registered for the same variable, and they fire in the order they were added.

Reaction Callbacks Take Exactly Two Parameters

The runtime always passes two arguments, so a one-parameter callback fails the moment it fires: `reaction error: expected 1 arguments but got 2` on the bytecode backends, `reaction error: closure expects 1 arguments, got 2` under `--tree-walk`. Write `|phase, val|` even when you only care about the phase.

And remember the lambda rule from Chapter 6: lambda parameters take no type annotations. `react(t, |phase: any| { ... })` is not a subtle phase bug, it is a parse error—`expected ',', got ':'`.

Under the hood, reactions are stored in the `LatRuntime` struct (see `include/runtime.h`). Each `RTReaction` entry associates a variable name with an array of callback closures. When `OP_FREEZE_VAR` or `OP_THAW_VAR` executes in the VM, it calls `stackvm_fire_reactions()` to invoke all registered callbacks.

Listing 13.11: Multiple reactions on one variable

```
flux sensor = 98.6

react(sensor, |phase, val| {
    print("Logger: sensor is now " + phase)
})

react(sensor, |phase, val| {
    if phase == "crystal" {
        print("Alert: sensor has been frozen!")
    }
})

freeze(sensor)
// Output: Logger: sensor is now crystal
// Output: Alert: sensor has been frozen!
```

13.3.2 Removing Reactions

`unreact()` removes all reaction callbacks registered on a variable:

Listing 13.12: Removing all reactions

```
flux value = 42
react(value, |phase, val| {
    print("phase changed: " + phase)
})

unreact(value)
freeze(value)  // No output---the callback has been removed
```

13.3.3 Bonds: Phase Dependency Relationships

While reactions let you *observe* phase changes, bonds let you *propagate* them. `bond()` creates a phase dependency: freezing the *first* variable drives a phase change in every variable named after it.

Listing 13.13: Creating a bond between variables

```
flux primary = [1, 2, 3]
flux replica = [1, 2, 3]

bond(primary, replica, "mirror")

// When primary freezes, replica automatically freezes too
freeze(primary)
print(phase_of(primary))  // "crystal"
print(phase_of(replica))  // "crystal" (frozen via cascade)
```

The direction is the thing to get right, and it reads left-to-right: `bond(source, dependent)` means "when `source` crystallizes, do something to `dependent`." Reverse the arguments and nothing happens at all—the cascade never fires, and there is no error to tell you so.

The trailing string is the *strategy*, which decides *what* happens to the dependent variables. Omit it and you get `"mirror"`. You can name several dependents in one call—`bond(primary, replica, backup, "mirror")`—and they all receive the same strategy.

Bond Requirements

Bonds have two requirements:

1. Every named variable must already exist: `cannot bond undefined variable 'replica'`.
2. The source variable must not already be frozen: `cannot bond already-frozen variable 'primary'`. A crystal value has no future transitions to broadcast.

The tree-walker is stricter than the bytecode backends here—it applies the not-already-frozen rule to the dependents as well, so a bond whose dependent is already crystal is rejected under `--tree-walk` but accepted on the VMs. Bond *before* you freeze anything and all three backends agree. (The `--regvm` wording differs slightly: `bond: variable 'primary' is already frozen`.)

Bonds Are One-Shot

A bond fires once and then dissolves. The runtime consumes the entire bond entry as part of the cascade, so freezing the source a second time propagates nothing:

```
flux a = 1
flux b = 2
bond(a, b, "mirror")

freeze(a)
print(phase_of(b))  // crystal

thaw(a)
thaw(b)
freeze(a)
print(phase_of(b))  // fluid --- the bond is gone
```

Bonds model initialization sequences and one-way sealing, not standing subscriptions. If you need a relationship that survives thawing, re-establish the bond after each thaw.

13.3.4 Removing Bonds

`unbond()` removes a specific dependent from a source:

Listing 13.14: Removing a bond

```
flux source = 100
flux follower = 100

bond(source, follower, "mirror")
unbond(source, follower)

freeze(source)
print(phase_of(follower))  // "fluid" (no longer bonded)
```

If removing a dependent leaves a bond entry with no dependents, the entire bond entry is cleaned up.

13.4 Bond Strategies: Mirror, Inverse, Gate

The strategy string in `bond()` controls what the cascade does to each dependent. The runtime recognizes exactly three: `"mirror"`, `"inverse"`, and `"gate"`. All three are handled by `rt_freeze_cascade()`

in `src/runtime.c`, which walks the bond table when a variable freezes, finds the entry whose source matches, and dispatches on the per-dependent strategy string stored in `RTBond.dep_strategies`.

13.4.1 Mirror: Freeze Follows Freeze

The `"mirror"` strategy is the default. When the source freezes, each mirrored dependent freezes too—and then cascades in turn, so chains propagate all the way down. Dependents that are already crystal are skipped.

Listing 13.15: Mirror bond strategy

```
flux leader = "hello"
flux follower = "world"

bond(leader, follower, "mirror")

freeze(leader)
print(phase_of(follower))  // "crystal" (mirrored freeze)
```

13.4.2 Inverse: Freeze Releases

The `"inverse"` strategy runs the transition backwards: freezing the source *thaws* a crystal dependent. This is how you hand off custody. Freezing the published copy of a document releases the draft for editing; a dependent that is already fluid is left alone.

Listing 13.16: Inverse bond strategy

```
flux draft = ["chapter one"]
flux published = []

bond(draft, published, "inverse")

freeze(published)        // published is sealed
freeze(draft)            // freezing draft releases published

print(phase_of(draft))      // "crystal"
print(phase_of(published))  // "fluid" --- thawed by the inverse cascade
```

Unlike mirror, inverse does not recurse: it thaws the dependent, fires its reactions, and stops.

13.4.3 Gate: Freeze Requires Freeze

The `"gate"` strategy is a precondition rather than a propagation. It changes nothing; it refuses to let the source freeze until the dependent is already crystal. Use it to enforce ordering—a schema must be sealed before the records that conform to it.

Listing 13.17: Gate bond strategy

```
flux schema = ["id", "name"]
flux records = [[1, "Aria"]]

bond(records, schema, "gate")

freeze(schema)   // gate satisfied
freeze(records)
print(phase_of(records))  // "crystal"
```

Freeze the records first and the gate slams shut:

Listing 13.18: A gate rejecting an out-of-order freeze

```
flux schema = ["id", "name"]
flux records = [[1, "Aria"]]

bond(records, schema, "gate")
freeze(records)
// error: gate bond: 'schema' must be crystal before 'records' can freeze
```

Bond Before You Freeze

Establish every bond while all the participants are still fluid. The gate example above works on all three backends only because `bond()` runs before either `freeze()`; bonding to an already-crystal dependent is rejected under `--tree-walk`. Bond first, freeze second, and the three backends behave identically.

13.5 Pressure: `pressurize()`, `depressurize()`, `pressure_of()`

Pressure is a constraint system for *structural mutations* on collections. While the phase system controls whether a value can be mutated at all, pressure controls *how* it can be mutated—specifically, whether a collection can grow, shrink, or both.

Listing 13.19: Applying pressure constraints

```
fn collect_log() {
    flux log_entries = ["boot", "init"]

    // Apply pressure: no shrinking allowed
    pressurize(log_entries, "no_shrink")

    log_entries.push("request received")  // OK: growing is allowed
    print(log_entries)  // [boot, init, request received]

    log_entries.pop()
    // error: pressurized (no_shrink): cannot pop from 'log_entries'
}
collect_log()
```

Like `bond()` and `react()`, `pressurize()` takes a *bare variable name*. Its sibling `depressurize()` does too. The odd one out is `pressure_of()`, which is an ordinary builtin and wants the name as a *string*.

Pressure Is Not Enforced on Globals Under `--regvm`

Notice that the collection above lives inside a function. That is deliberate: the register VM checks pressure constraints against a frame's named locals, and a collection declared at the top level of a script is not a local. A pressurized *global* is therefore unguarded under `--regvm`, while the default stack VM and the tree-walker both reject the operation:

```
flux items = [1, 2, 3]
pressurize(items, "no_grow")
items.push(4)     // errors on the stack VM and --tree-walk
                  // silently succeeds on --regvm
```

Keep pressurized collections function-scoped and all three backends agree. Treat pressure as a design aid rather than a security boundary—phase, not pressure, is the mechanism with real teeth.

13.5.1 Pressure Modes

The `pressurize()` function accepts a variable name and a mode string. Four modes are available:

Mode	Effect
`"no_grow"`	Blocks `push()`, `insert()`, and other size-increasing operations.
`"no_shrink"`	Blocks `pop()`, `remove_at()`, and other size-reducing operations.
`"no_resize"`	Blocks both growing and shrinking—the collection's size is locked.
`"read_heavy"`	An optimization hint indicating the collection will be read much more often than written. It constrains nothing—growth and shrinkage both proceed.

Table 13.1: Pressure modes and their effects.

Listing 13.20: Different pressure modes

```
fn rescore() {
    flux scores = [85, 92, 78, 95]

    // Lock the size entirely
    pressurize(scores, "no_resize")

    scores[0] = 90   // OK: element modification is still allowed
    print(scores)    // [90, 92, 78, 95]

    scores.push(100)
    // error: pressurized (no_resize): cannot push to 'scores'
}
rescore()
```

An unrecognized mode string is not an error you can count on: the tree-walker rejects it (`pressurize() unknown mode 'no_such_mode'`) while the bytecode backends discard the call and leave the collection unconstrained. Stick to the four names in Table 13.1.

Pressure vs. Phase

Pressure and phase are orthogonal. A fluid value under `"no_resize"` pressure can still have its *elements* modified—you just cannot add or remove elements. A crystal value rejects *all* modifications regardless of pressure. Think of pressure as a structural constraint (shape of the container) and phase as a material constraint (mutability of the contents).

13.5.2 Querying and Removing Pressure

Use `pressure_of()` to check the current pressure on a variable, and `depressurize()` to remove it:

Listing 13.21: Querying and removing pressure

```
flux items = [1, 2, 3]

pressurize(items, "no_grow")
print(pressure_of("items"))  // no_grow

depressurize(items)
print(pressure_of("items"))  // nil (no pressure)

items.push(4)  // Now allowed again
print(items)   // [1, 2, 3, 4]
```

Note the asymmetry once more: `pressurize` and `depressurize` take `items`, while `pressure_of` takes `"items"`. Registration is uniform across the backends even where enforcement is not, so `pressure_of` reports the mode on all three.

13.5.3 Implementation Details

Pressure constraints are stored in the `LatRuntime` struct as an array of `RTPressure` entries (see `include/runtime.h`). Each entry maps a variable name to a mode string. The pressure check is performed by array mutating methods in the VM (e.g., during `push()`, `pop()`, `insert()`, and `remove_at()`), which look up the variable name in the pressure table and reject operations that violate the constraint.

The native functions `native_pressurize()`, `native_depressurize()`, and `native_pressure_of()` are defined in `src/runtime.c`. They validate the mode string, update the pressure table, and handle the swap-remove pattern for deregistration.

13.6 Seeds: seed(), unseed()

Seeds are *validation contracts* that guard the crystallization process. When you seed a variable, you attach a closure that must approve the value before it can be frozen. A seed contract is a *predicate*: return something truthy and the crystal forms; return `false` or `nil` and the freeze is refused.

Listing 13.22: Setting a seed contract

```
flux user_age = 25

seed(user_age, |val| {
    return val >= 0 && val <= 150
})

// This freeze works: 25 passes the contract
fix sealed_age = freeze(user_age)
print(sealed_age)  // 25
```

Listing 13.23: Seed contract rejecting a freeze

```
flux invalid_score = -10

seed(invalid_score, |val| {
    return val >= 0
})

fix bad = freeze(invalid_score)
// error: grow() seed contract returned false
```

Seed

A *seed* is a validation contract (a closure) attached to a variable. The seed is checked when the variable is frozen via `freeze()` or `grow()`. If the closure returns a truthy value, the freeze proceeds. If it returns `false`, `nil`, or anything else falsy, the freeze is rejected and the variable stays fluid.

Seeds Approve, They Do Not Report

A seed contract answers "may this value crystallize?"—it is not a place to compose an error message. Returning a string like `"age cannot be negative"` is the *opposite* of what you mean: a non-empty string is truthy, so the freeze sails through. And a contract that returns `nil` on success rejects every value it was written to accept.

The rejection message is fixed, not something the contract supplies:

- `grow() seed contract returned false` — on the bytecode backends for both `freeze()` and `grow()`, and under `--tree-walk` for `grow()`.
- `seed contract failed on freeze: contract returned false` — under `--tree-walk` when a `freeze()` is what tripped it.

If a contract needs to explain itself, `print()` the reason before returning `false`.

13.6.1 Seeds and `grow()`

The `grow()` function is a higher-level operation that validates all seeds and then freezes the variable. Unlike a bare `freeze()`, `grow()` *consumes* the seed contracts—after a successful `grow()`, the seeds are removed:

Listing 13.24: Using grow() with seeds

```
flux balance = 1000

seed(balance, |val| { return val >= 0 })

// grow() validates seeds, freezes, and removes the seed contracts
fix final_balance = grow("balance")
print(final_balance)      // 1000
print(phase_of(balance))  // crystal
```

Note the argument: `seed()` and `unseed()` take a bare variable name, but `grow()` is an ordinary builtin and wants the name as a string—the same split as `pressurize()` versus `pressure_of()`.

The `grow()` function is implemented as `native_grow()` in `src/runtime.c`. It calls `rt_validate_seeds()` with the `consume` flag set to `true`, which both validates the contracts and removes them from the runtime's seed table upon success. After validation, it freezes the value, records the phase change in the tracking history, fires any cascade bonds, and triggers reactions.

A plain `freeze()` calls the same validator with `consume` set to `false`, which is the whole difference between the two: `freeze()` consults the contracts and leaves them in place, `grow()` consults them once and burns them. After a `grow()`, a later `freeze()` of the same variable is unguarded.

13.6.2 Removing Seeds Manually

`unseed()` removes the seed contract for a variable without triggering any validation:

Listing 13.25: Removing a seed contract

```
flux data = "sensitive"

seed(data, |val| { return len(val) >= 8 })

// Changed our mind---remove the seed
unseed(data)

// Now freeze works without validation
fix sealed = freeze(data)
print(sealed)  // sensitive
```

13.6.3 Multiple Seeds

You can attach multiple seed contracts to the same variable. All of them must approve the value for the freeze to succeed:

Listing 13.26: Multiple seed contracts

```
flux password = "hunter2"

seed(password, |val| { return len(val) >= 8 })
seed(password, |val| { return val != "password" })

// "hunter2" is only 7 characters---the length seed rejects it
fix sealed = freeze(password)
// error: grow() seed contract returned false
```

13.7 Building Reactive Data Flows

By combining bonds, reactions, seeds, and pressure, you can build sophisticated reactive data flows where phase transitions cascade through your data model automatically.

13.7.1 Example: Configuration Validation Pipeline

Here is a complete example that uses seeds to validate configuration, bonds to cascade freezing from a primary config to dependent caches, and reactions to log all phase changes:

Listing 13.27: A reactive configuration pipeline

```
// Set up the primary configuration
flux app_config = Map::new()
app_config.set("database_url", "postgres://localhost/mydb")
app_config.set("cache_ttl", "300")
app_config.set("max_retries", "3")

// Set up a dependent cache configuration
flux cache_config = Map::new()
cache_config.set("ttl", "300")
cache_config.set("backend", "redis")

// Bond: when app_config freezes, cache_config freezes too
bond(app_config, cache_config, "mirror")

// Seed: validate configuration before freezing
seed(app_config, |val| {
    return val.get("database_url") != nil
})

// Reaction: log phase changes
react(app_config, |phase, val| {
    print("app_config phase -> " + phase)
})

react(cache_config, |phase, val| {
    print("cache_config phase -> " + phase)
})

// Freeze the primary config
// 1. Seed validates (database_url exists) -> passes
// 2. app_config freezes
// 3. Bond cascade: cache_config freezes, its reaction fires
// 4. app_config's own reaction fires last
fix sealed_config = freeze(app_config)
// Output: cache_config phase -> crystal
// Output: app_config phase -> crystal

print(phase_of(app_config))    // crystal
print(phase_of(cache_config))  // crystal
```

That ordering is worth pausing on. The cascade runs *before* the source variable's own reactions, so the dependents announce themselves first and the variable you actually froze reports last. Chains deepen the effect: the far end of the chain speaks before the near end.

13.7.2 Example: Sensor Data with Pressure Guards

Listing 13.28: Sensor data with pressure and seeds

```
flux readings = []

// Pressure: this buffer only grows (no accidental data loss)
pressurize(readings, "no_shrink")

// Seed: require at least 10 readings before allowing freeze
seed(readings, |val| {
    return len(val) >= 10
})

// Collect sensor data
for i in 0..15 {
    readings.push(20.0 + to_float(i) * 0.5)
}

// readings.pop()
// error: pressurized (no_shrink): cannot pop from 'readings'

// Now we have 15 readings, seed contract will pass
fix sealed_readings = freeze(readings)
print(sealed_readings.len())  // 15
```

13.7.3 Example: Multi-Stage Initialization

Listing 13.29: Multi-stage initialization with bonds

```
flux database = Map::new()
flux http_server = Map::new()
flux app = Map::new()

// Set up cascading bonds
bond(database, http_server, "mirror")
bond(http_server, app, "mirror")

// Reactions for monitoring
react(database, |phase, val| {
    print("Database: " + phase)
})
react(http_server, |phase, val| {
    print("HTTP Server: " + phase)
})
react(app, |phase, val| {
    print("Application: " + phase)
})

// Configure each component
database.set("host", "localhost")
database.set("port", "5432")

http_server.set("port", "8080")
http_server.set("workers", "4")

app.set("name", "MyApp")
app.set("version", "1.0.0")

// Freeze the database layer
// This cascades: database -> http_server -> app
freeze(database)
// Output:
//   HTTP Server: crystal
//   Application: crystal
//   Database: crystal

print(phase_of(app))  // crystal
```

Cascade Ordering

Bond cascades propagate synchronously and *depth-first* during the freeze operation. Freezing A in a chain A → B → C freezes B, fires B's reactions, recurses to C, fires C's reactions—and only then returns to fire A's own reactions. That is why the log above reads bottom-up.
Be careful with long chains: each step is a full freeze, which materializes the value into a crystal region. And every bond in the chain is consumed as it fires, so a second pass through the same initialization sequence propagates nothing.

13.8 Exercises

1. **Alloy Design.** Design an alloy struct for a bank account with immutable fields for the account number and holder name, and mutable fields for the balance and transaction history. Write functions to deposit, withdraw, and display the account. Ensure the account number can never be changed.

2. **Reactive Counter.** Create two variables, `counter` and `display`. Set up a reaction on each that prints the phase change, then bond them with the mirror strategy so that freezing `counter` freezes `display`. Get the argument order right on the first try, then deliberately reverse it and observe that the cascade silently does nothing. Which of the two reactions fires first?

3. **Seed Validation.** Write a seed contract for an email address variable that approves values containing an `@` character and at least 5 characters. Test it by freezing valid and invalid addresses. Then break it on purpose: make the contract return a descriptive string on failure instead of `false`, and explain why every value now passes.

4. **Pressure Experiment.** Inside a function, create an array under `"no_grow"` pressure. Verify that `push()` fails but element assignment works. Then depressurize and confirm `push()` works again. Switch to `"no_resize"` and verify both `push()` and `pop()` fail. Run it on all three backends; then hoist the array to the top level and run it again.

5. **Strategy Tour.** Build three pairs of variables and bond each pair with a different strategy—`"mirror"`, `"inverse"`, and `"gate"`. Predict the phase of every variable after freezing each source, then check your predictions with `phase_of()`. Trigger the gate error by freezing in the wrong order.

6. **Complete Pipeline.** Build a complete reactive pipeline with three variables: `raw_data`, `validated_data`, and `archived_data`. Use seeds to validate `raw_data` before freezing, bonds to cascade the freeze to `validated_data` and then `archived_data`, and reactions to log every phase transition. Then freeze a second time and explain the silence.

What's Next. We have explored the full range of Lattice's phase system: from basic `flux`/`fix` declarations through alloy structs, reactive bonds, and validation seeds. All of it has operated in *casual* mode, where the runtime catches phase violations as they happen—and since v0.5.0 it really does catch them, on every backend and every mutation path. That is a genuine safety net, but it only tightens when the offending line executes. In Chapter 14, we explore strict mode and the static phase checker—an analysis that runs before the first instruction and reports every phase error in the file at once.

Chapter 14

Strict Mode and Static Phase Checking

Throughout this part of the book, we have explored the phase system as a runtime concept: values carry phase tags, `freeze()` changes them, and—since v0.5.0—every backend refuses every mutation of a crystal value. But runtime checks, by definition, happen *while your program is running*. A phase violation in rarely-executed code might lurk undetected for months.

Lattice offers a way to catch these errors earlier: *strict mode*. By adding `#mode strict` at the top of a file, you activate a static phase checker that analyzes your code *before* execution and rejects programs that violate phase rules. Strict mode does not eliminate runtime phase checks—it adds a layer of compile-time verification on top of them.

In this chapter, we explore what strict mode changes, how the static phase checker works, how to use phase annotations and constraints, and how to design APIs that take full advantage of phase-aware overloading.

14.1 `#mode strict` — What Changes

To enable strict mode, add a mode directive at the very beginning of your file:

Listing 14.1: Enabling strict mode

```
#mode strict

flux counter = 0
fix name = freeze("Lattice")
```

The directive is processed by the lexer (`src/lexer.c`) as a `TOK_MODE_DIRECTIVE` token and stored in the program's AST as the `AstMode` field (either `MODE_CASUAL` or `MODE_STRICT`, defined in `include/ast.h`). If no directive is present, the default is `MODE_CASUAL`; `#mode casual` states it explicitly.

The word "beginning" is literal. The directive is a statement like any other, and the parser only accepts it in the first position:

Listing 14.2: A misplaced directive

```
flux a = 1
#mode strict
// error: 2:1: parse error: unexpected token '#mode' in expression
```

When strict mode is active, the following rules are enforced by the static phase checker before any code executes. Every message below is reported with a `phase error:` prefix—distinct from the `vm error:`/`error:`/`regvm error:` prefixes you have seen so far, because nothing has run yet. The messages are byte-identical on all three backends: the checker sits in front of the compilers, not inside them.

1. **No `let` bindings.** Every variable must use `flux` or `fix`. The phase checker rejects `let` with:

   ```
   strict mode: use 'flux' or 'fix' instead of 'let' for binding 'x'
   ```

2. **No assigning to crystal bindings.** If a variable was declared with `fix` (or its expression was determined to be crystal), assignment to it is an error:

   ```
   strict mode: cannot assign to crystal binding 'config'
   ```

3. **No redundant freeze or thaw.** Freezing an already-crystal value or thawing an already-fluid value is flagged:

   ```
   strict mode: cannot freeze an already crystal value
   strict mode: cannot thaw an already fluid value
   ```

4. **No phase mismatch on binding.** Binding a crystal expression with `flux` is an error:

   ```
   cannot bind crystal value with flux for 'x'
   ```

5. **No fluid values in `spawn`.** Referencing a fluid variable inside a `spawn` block is rejected:

   ```
   strict mode: cannot use fluid binding 'counter' across thread boundary in
   spawn
   ```

6. **Destructuring requires explicit phase.** Destructuring bindings must specify `flux` or `fix`:

   ```
   strict mode: destructure requires explicit phase (flux/fix)
   ```

So `fix [a, b] = pair` is accepted where `let [a, b] = pair` is not.

7. **Phase annotations are validated.** `@fluid` and `@crystal` annotations are checked against the initializer's inferred phase.

8. **Function arguments checked against parameter phases.** If a function declares parameter phases, the checker verifies that call-site arguments have compatible phases.

The checker does not stop at the first complaint. It walks the whole file and prints everything it found:

Listing 14.3: Every phase error at once

```
#mode strict
let a = 1
let b = 2
fix c = freeze(3)
c = 4
// phase error: strict mode: use 'flux' or 'fix' instead of 'let' for binding 'a'
// phase error: strict mode: use 'flux' or 'fix' instead of 'let' for binding 'b'
// phase error: strict mode: cannot assign to crystal binding 'c'
```

14.1.1 Consume on Freeze

Strict mode carries one rule that the phase checker has nothing to do with, because it happens while the program runs rather than before it: `freeze()` is meant to *consume* the binding it froze. The idea is a good one. If you freeze a value and hand the crystal to somebody else, the molten name you used during construction has no business still being in scope—so strict mode takes it away, and you use the new binding from then on.

Listing 14.4: The shape strict-mode freezing is meant to have

```
#mode strict

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

fix frozen = freeze(data)
// `frozen` is the name you use from here on
print(frozen)  // [1, 2, 3, 4]
```

Consume-on-Freeze Lives Only in the Tree-Walker

As of v0.6.4 this rule is implemented in exactly one backend. The tree-walking interpreter removes the binding (`src/eval.c` calls `env_remove` on the strict-mode `freeze` path); the default stack VM and `--regvm` leave the name in place. Append `print(data)` to the listing above and you get:

```
// --tree-walk:  error: undefined variable 'data'
// default VM:   [1, 2, 3, 4]
// --regvm:      [1, 2, 3, 4]
```

The *value* is crystal on all three—that part never diverges. Write strict-mode code as though the name really does vanish and it will be correct everywhere; do not write code that depends on either answer.

Strict Mode Is Per-File—and Only the Entry File Is Checked

The `#mode strict` directive applies to the file where it appears. You can have strict-mode library code called by casual-mode application code, or vice versa.

There is a wrinkle worth knowing. The static checker runs on the program you hand to `clat`, and only on that program. A module you `import` is compiled at runtime, and its own `#mode strict` directive does not summon the checker—so a library can carry strict-mode violations that only surface when someone runs the file directly. Under `--tree-walk` there is an additional wrinkle in the other direction: a strict-mode *importer* propagates its mode into imported code, and the same library that runs fine on the bytecode backends raises a runtime error there.

If you write strict-mode libraries, run them through `clat` directly (or their test suite) rather than trusting the importer to check them.

14.2 The Static Phase Checker

The static phase checker is implemented in `src/phase_check.c`. It is a single-pass abstract interpreter that walks the AST, tracking the phase of every binding in a scope stack, and emitting errors when phase rules are violated.

14.2.1 Architecture

The checker maintains a `PhaseChecker` struct with:

- `mode` — the `AstMode` (`MODE_CASUAL` or `MODE_STRICT`).
- `errors` — a vector of error message strings.

- `scope_stack` — a stack of hash maps (`LatMap`), each mapping variable names to their `AstPhase` (the compile-time phase annotation, not the runtime tag).
- `struct_defs` — a map of struct declarations for looking up field types.
- `fn_defs` — a map of function declarations for checking call-site argument phases.

14.2.2 Scope Tracking

The checker uses a scope stack to track variable phases through nested blocks. When a new scope begins (function body, loop body, if-branch, block expression), `pc_push_scope()` creates a fresh map. When the scope ends, `pc_pop_scope()` discards it. Variable lookup (`pc_lookup()`) searches from the innermost scope outward, mimicking runtime variable resolution:

Listing 14.5: Phase tracking through scopes

```
#mode strict

flux outer = 10

if outer > 5 {
    fix inner = freeze(outer)
    // Phase checker knows: outer=fluid, inner=crystal
    // inner = 20  // Error: cannot assign to crystal binding
}
// Phase checker knows: inner is out of scope now
```

14.2.3 Expression Phase Inference

The function `pc_check_expr()` recursively analyzes each expression and returns its inferred phase:

- **Literals** (integers, floats, strings, booleans, nil): return `PHASE_UNSPECIFIED`.
- **Identifiers**: return the phase registered in the scope stack via `pc_lookup()`.
- **Freeze expressions**: check the inner expression, then return `PHASE_CRYSTAL`. In strict mode, if the inner expression is already crystal, emit an error.
- **Thaw expressions**: return `PHASE_FLUID`. In strict mode, if the inner expression is already fluid, emit an error.
- **Clone expressions**: return the same phase as the inner expression.
- **Tuple expressions**: always return `PHASE_CRYSTAL` (tuples are inherently immutable).

- **Field access**: if the object is crystal, the accessed field is also crystal.
- **Forge blocks**: always return `PHASE_CRYSTAL`.
- **Anneal expressions**: always return `PHASE_CRYSTAL`.
- **Function and method calls**: always return `PHASE_UNSPECIFIED`.

That last one deserves a note, because it explains a great deal of the checker's temperament. The checker never infers a phase *through* a call. It does not matter that a function is annotated `@crystal` or that its body ends in `freeze()`—at the call site, the result is unspecified, and unspecified is compatible with everything. So `fix config = build_config(host, port)` passes, and so does `flux config = build_config(host, port)`. The checker reasons about what it can see locally and declines to guess about the rest.

The same conservatism shows up in the other direction. Literals are unspecified too, which is why `@crystal flux counter = 0` draws no complaint: the checker knows nothing about `0` that contradicts the annotation. Bind the same annotation to a fluid *identifier* and it fires—Section 14.3 walks through that example and what an annotation does and does not promise.

14.2.4 Statement Checking

The function `pc_check_stmt()` handles bindings, assignments, loops, and other statements:

- **Bindings**: In strict mode, `PHASE_UNSPECIFIED` (i.e., `let`) triggers an error. The phase of the binding is recorded: `PHASE_FLUID` for `flux`, `PHASE_CRYSTAL` for `fix`. Cross-phase binding (e.g., crystal value into `flux` binding) is flagged.
- **Assignments**: In strict mode, if the target identifier is crystal, the assignment is rejected.
- **Import statements**: Imported bindings are registered as `PHASE_UNSPECIFIED`.

14.2.5 The Spawn Checker

Strict mode includes a specialized checker for `spawn` blocks: `pc_check_spawn_expr()` and `pc_check_spawn_stmt()` in `src/phase_check.c`. These functions walk the spawn body and flag any reference to a fluid binding from an outer scope:

Listing 14.6: The spawn checker in action

```
#mode strict

flux shared = [1, 2, 3]
fix safe = freeze([4, 5, 6])

scope {
    spawn {
        // print(shared)  // Error: cannot use fluid binding
        //                //        'shared' in spawn
        print(safe)        // OK: safe is crystal
    }
}
```

This check catches potential data races at compile time. In casual mode, the same code would run and might produce incorrect results if the fluid value were modified concurrently. Strict mode prevents this entire class of bugs.

14.3 Phase Annotations: `@fluid`, `@crystal`

Phase annotations are optional markers that assert the expected phase of a binding or function. They are placed before the declaration using the `@` symbol:

Listing 14.7: Phase annotations on variable bindings

```
#mode strict

@crystal
fix config = freeze(Map::new())

@fluid
flux counter = 0
```

14.3.1 Annotation Syntax

The parser recognizes two annotations: `@fluid` and `@crystal`. They can appear before:

- **Variable bindings**: `@crystal fix x = freeze(42)`

- **Function declarations**: `@fluid fn process(data: any) { ... }`

Any other annotation name (e.g., `@readonly`, `@mutable`) produces a parse error:

```
unknown annotation '@readonly' (expected @fluid or @crystal)
```

If an annotation appears before a construct that is not a function or binding, the parser also rejects it:

```
@fluid/@crystal annotation must precede fn or variable binding
```

14.3.2 What Annotations Do

Annotations are *assertions*, not directives. They do not change the phase of the value—they verify that the value's phase matches the annotation. The phase checker validates annotations against the initializer's inferred phase:

Listing 14.8: Annotation violation

```
#mode strict

flux counter = 0
@crystal
flux mirror = counter
// phase error: @crystal annotation violated: initializer for 'mirror' is fluid
```

The initializer has to be an *identifier* for the check to have anything to inspect. A literal carries no phase of its own, so `@crystal flux counter = 0` draws no complaint at all—the checker sees UNSPECIFIED, not FLUID. Section 14.2 works through why.

The reverse also triggers an error, and there a literal initializer is enough, because `freeze()` stamps a phase onto its result:

Listing 14.9: Reverse annotation violation

```
#mode strict

@fluid
fix data = freeze([1, 2, 3])
// phase error: @fluid annotation violated: initializer for 'data' is crystal
```

14.3.3 Annotations on Functions

When applied to a function, a phase annotation indicates the expected phase behavior of the function's return value or its overall role in the phase system:

Listing 14.10: Annotated functions

```
#mode strict

@crystal
fn default_config() {
    return freeze(Map::new())
}

@fluid
fn create_buffer(size: any) {
    flux buf = []
    for i in 0..size {
        buf.push(0)
    }
    return buf
}
```

These annotations serve as documentation and contract: they signal to readers (and to the phase checker) the function's intended phase behavior.

Annotations as Documentation

Even in casual mode, phase annotations are valuable as documentation. They make the author's intent explicit without requiring strict mode's full enforcement. Consider adding them to public API functions in library code.

14.4 Phase Constraints: `(~| *)`, `(flux| fix)`

While annotations assert a single phase, *phase constraints* on function parameters specify which phases are acceptable for an argument. Constraints use a compact syntax inside parentheses, with the pipe symbol `|` separating alternatives:

Listing 14.11: Phase constraints on parameters

```
#mode strict

fn display_value(data: (~|*) any) {
    // data can be either fluid (~) or crystal (*)
    print(data)
}

fn read_only_process(data: (*) any) {
    // data must be crystal
    print("Processing: " + to_string(data))
}
```

14.4.1 Constraint Syntax

Phase constraints are parsed by the type expression parser in `src/parser.c`. Inside parentheses, the following symbols are recognized:

Symbol	Meaning
`~` or `flux`	Accepts fluid values (`PCON_FLUID`)
`*` or `fix`	Accepts crystal values (`PCON_CRYSTAL`)
`sublimated`	Accepts sublimated values (`PCON_SUBLIMATED`)

Table 14.1: Phase constraint symbols.

Constraints are represented internally as a bitmask (`PhaseConstraint` type, defined in `include/ast.h`):

- `PCON_FLUID` = `0x01`
- `PCON_CRYSTAL` = `0x02`
- `PCON_SUBLIMATED` = `0x04`
- `PCON_ANY` = `0x07` (all bits set)

Multiple symbols are combined with the pipe (`|`) separator:

Listing 14.12: Composite phase constraints

```
#mode strict

// Accept fluid or crystal, but not sublimated
fn flexible_handler(val: (~|*) any) {
    print(to_string(val))
}

// Accept only crystal or sublimated (immutable variants)
fn immutable_handler(val: (*|sublimated) any) {
    print(to_string(val))
}
```

14.4.2 Constraint Checking

The function `constraint_accepts()` in `src/phase_check.c` checks whether a given argument phase satisfies a constraint bitmask:

- If the constraint is 0 (no constraint specified), any phase is accepted.
- Otherwise, the argument's phase must have its corresponding bit set in the bitmask.
- `PHASE_UNSPECIFIED` arguments are always accepted (they are compatible with any constraint).

In strict mode, the phase checker verifies constraints at every call site:

Listing 14.13: Constraint violation in strict mode

```
#mode strict

fn process_frozen(data: (*) any) {
    print(data)
}

flux mutable_data = [1, 2, 3]
// process_frozen(mutable_data)
// Error: strict mode: no matching overload for 'process_frozen'
//        with given argument phases
```

Phase Constraint

A *phase constraint* is a bitmask attached to a function parameter's type that specifies which value phases are acceptable for that argument. Constraints are checked at compile time in strict mode and during overload resolution at runtime.

Which Spellings Survive Into Casual Mode

The two syntaxes are not equally load-bearing once the checker goes away. The *prefix* forms—`~T`, `*T`, and their long spellings `flux T` and `fix T`—are enforced at every call, in casual mode as well as strict. The *parenthesized* composite forms, `(~|*) T` and friends, are a strict-mode contract only: the static checker tests them and nothing checks them at runtime. So `fn only_frozen(data: (*) any)` accepts a fluid array without complaint in casual mode, while `fn inspect(data: *Map)` refuses one.

The refusal wording is one of the places the backends still diverge:

- Default stack VM and `--regvm`: `phase constraint violation in function 'inspect'`
- `--tree-walk`: `function 'inspect' parameter 'data' requires fix argument, got fluid (flux)`

Match on the function name, not on the rest of the sentence.

14.5 Phase-Dependent Function Overloading

One of Lattice's most distinctive features is *phase-dependent function overloading*: you can define multiple versions of a function with the same name but different parameter phase constraints, and the runtime (or the strict-mode checker) selects the appropriate one based on the phases of the actual arguments.

Listing 14.14: Phase-dependent overloading

```
fn process(data: flux any) {
    print("Processing mutable data in-place")
    data.push(0)
}

fn process(data: fix any) {
    print("Processing frozen data (creating a copy)")
    flux copy = thaw(data)
    copy.push(0)
    return freeze(copy)
}

flux live_data = [1, 2, 3]
fix frozen_data = freeze([4, 5, 6])

process(live_data)    // Calls the flux overload
process(frozen_data)  // Calls the fix overload
```

14.5.1 How Overload Resolution Works

Functions with the same name are chained together in a linked list via the `next_overload` pointer in the `FnDecl` struct (`include/ast.h`). When a function is called, the runtime walks this chain to find the best matching overload.

The `resolve_overload()` function in `src/eval.c` implements the resolution algorithm:

1. For each overload candidate, check **arity compatibility**: does the argument count match the parameter count (accounting for default values and variadic parameters)?
2. For each argument, check **phase compatibility**: is the argument's runtime phase compatible with the parameter's declared phase?
3. Compute a **score** for the match. Exact phase matches score highest (3 points each), compatible-but-unspecified matches score lower (1–2 points).
4. Select the candidate with the highest total score. If no candidate is compatible, emit an error:

```
no matching overload for 'process' with given argument phases
```

The scoring system ensures that more specific overloads are preferred. An overload declaring `flux` for a parameter will beat one declaring no phase (unspecified) when passed a fluid argument.

14.5.2 Overloading in Strict Mode

In strict mode, the phase checker performs a compile-time version of overload resolution. When a call expression targets a function name that has multiple overloads, the checker:

1. Infers the phase of each argument expression.
2. Tests each overload's parameter constraints against the inferred argument phases.
3. If no overload matches, emits:

```
strict mode: no matching overload for 'process' with given argument phases
```

This means phase-mismatch errors are caught before the program runs:

Listing 14.15: Compile-time overload checking

```
#mode strict

fn handle(data: flux any) {
    data.push(0)
}

fix frozen = freeze([1, 2, 3])
// handle(frozen)
// Error: strict mode: no matching overload for 'handle'
//        with given argument phases
```

14.5.3 Registration and Chaining

When the evaluator or compiler encounters multiple function declarations with the same name, it registers them using `register_fn_overload()` (in `src/eval.c` for the tree-walker, and `pc_register_fn()` in `src/phase_check.c` for the static checker).

Two functions with the same name are considered different overloads if their *phase signatures* differ. The function `phase_signatures_match()` compares parameter counts and per-parameter phases:

Listing 14.16: Different phase signatures create overloads

```
// These are two different overloads:
fn serialize(data: flux any) { /* ... */ }
fn serialize(data: fix any) { /* ... */ }

// These have the SAME phase signature (both unspecified):
fn parse(input: any) { /* version 1 */ }
fn parse(input: any) { /* version 2: replaces version 1 */ }
```

If two functions have identical phase signatures, the second definition *replaces* the first (it is not treated as a new overload). Only functions with *different* phase signatures coexist in the overload chain.

Overload Ambiguity

If an argument is `VTAG_UNPHASED` (e.g., from a `let` binding in casual mode), it is compatible with both fluid and crystal overloads. The resolver picks the one with the highest score, but if scores are tied, the last-registered overload wins. In strict mode, `let` is forbidden, so unphased ambiguity does not arise.

14.6 Designing APIs That Are Phase-Aware

The phase system is most powerful when your API boundaries are designed with phases in mind. Here are patterns and guidelines for building phase-aware libraries and modules.

14.6.1 Produce Crystal, Accept Any

A common pattern: functions that return data should return it frozen (crystal), while functions that accept data should be flexible about the input phase:

Listing 14.17: Produce crystal, accept any

```
fn get_defaults() {
    // Always return frozen defaults
    return forge {
        flux temp = Map::new()
        temp.set("timeout", "30")
        temp.set("retries", "3")
        freeze(temp)
    }
}

fn apply_config(base: any, overrides: any) {
    // Accept any phase, thaw if needed
    flux working = thaw(base)
    // Apply overrides...
    return freeze(working)
}
```

14.6.2 Phase-Specific Behavior via Overloading

Use overloading to provide optimized paths for different phases:

Listing 14.18: Optimized paths via phase overloading

```
fn update_score(scores: flux Array, index: any, delta: any) {
    // Fluid path: modify in place (efficient)
    scores[index] = scores[index] + delta
    return scores
}

fn update_score(scores: fix Array, index: any, delta: any) {
    // Crystal path: create modified copy
    flux copy = thaw(scores)
    copy[index] = copy[index] + delta
    return freeze(copy)
}
```

Callers do not need to know which overload is selected—the function handles both cases correctly.

14.6.3 Guard Clauses with `phase_of()`

For functions that need phase-dependent behavior in casual mode (where overloading may not be strictly enforced), use runtime phase checks as guard clauses:

Listing 14.19: Runtime phase guards

```
fn safe_update(collection: any, key: any, value: any) {
    if phase_of(collection) == "crystal" {
        print("Warning: cannot update crystal collection")
        return collection
    }
    if phase_of(collection) == "sublimated" {
        print("Error: collection is permanently sealed")
        return collection
    }
    collection.set(key, value)
    return collection
}
```

14.6.4 Strict Mode in Library Code

Consider enabling strict mode in library code even if your application code uses casual mode. Libraries are consumed by many callers, and phase errors in library code can be particularly hard to debug:

Listing 14.20: Strict mode for library safety

```
// lib/cache.lat
#mode strict

@crystal
fn create_cache(max_size: any) {
    return forge {
        flux temp = Map::new()
        temp.set("__max_size", to_string(max_size))
        freeze(temp)
    }
}

fn cache_get(cache: fix any, key: any) {
    return cache.get(key)
}
```

14.6.5 Documenting Phase Expectations

Even without strict mode, use comments and naming conventions to signal phase expectations:

Listing 14.21: Phase documentation conventions

```
// Convention: functions returning frozen data end with _frozen
fn load_config_frozen(path: any) {
    // ... load and return frozen config ...
    return freeze(config)
}

// Convention: functions taking mutable data start with mutate_
fn mutate_append(list: any, item: any) {
    list.push(item)
}
```

Phase-Aware API Checklist

When designing a public API, ask these questions:

1. Should returned values be frozen? (Usually yes, for safety.)
2. Do different parameter phases warrant different behavior? (Consider overloading.)
3. Should the function reject certain phases? (Use constraints or runtime guards.)
4. Is this code safety-critical enough for strict mode? (Libraries: probably yes.)

14.7 Putting It All Together: A Strict-Mode Example

Let us write a complete strict-mode program that demonstrates phase annotations, constraints, overloading, and the spawn checker:

Listing 14.22: A complete strict-mode program

```
#mode strict

// Phase-annotated configuration builder
@crystal
fn build_config(host: any, port: any) {
    return forge {
        flux temp = Map::new()
        temp.set("host", host)
        temp.set("port", to_string(port))
        temp.set("started_at", "2024-01-15T10:00:00Z")
        freeze(temp)
    }
}

// Phase-dependent overloads for processing
fn process_request(config: fix any, path: any) {
    fix response = freeze("200 OK: " + path)
    return response
}

// Entry point
fn main() {
    fix config = build_config("0.0.0.0", 8080)

    // config is crystal---safe to share across spawn boundaries
    fix paths = freeze(["/api/health", "/api/users", "/api/data"])

    scope {
        for path in paths {
            spawn {
                // All bindings here are crystal---strict mode is happy
                fix result = process_request(config, path)
                print(result)
            }
        }
    }
}
```

This program compiles without any strict-mode errors because:

- Every binding uses `flux` or `fix` (no `let`).

- Only crystal values (`config`, `paths`) cross the spawn boundary.
- The `process_request` function's parameter constraint matches the crystal argument.
- Phase annotations on `build_config` match its forge-block return (crystal).

14.8 Exercises

1. **Strict Mode Conversion.** Take any program you have written while following this book and add `#mode strict` at the top. Fix all the phase errors the checker reports. How many `let` bindings did you have to change to `flux` or `fix`?
2. **Phase-Overloaded API.** Write a function `sort_data` with two overloads: one for fluid arrays (sorts in place) and one for crystal arrays (returns a new sorted frozen array). Test both overloads and verify the correct one is called.
3. **Constraint Design.** Write a function that accepts only crystal or sublimated values using the `(*sublimated)|` constraint syntax. Test it with fluid, crystal, and sublimated arguments. Verify that the fluid argument is rejected in strict mode.
4. **Spawn Safety.** Write a strict-mode program with a scope containing multiple spawn blocks. Try to reference a fluid variable from within a spawn and observe the compile-time error. Then fix the program by freezing the shared data before the scope.

What's Next. With the phase system fully explored—from basic `flux`/`fix` declarations through memory arenas, alloy structs, reactive bonds, and strict-mode static checking—we are ready to move beyond values and into the world of composite types. In Part IV, we explore how Lattice's struct, enum, trait, and generic systems interact with the phase system you now understand, giving you the tools to build sophisticated, type-safe abstractions.

Part IV

Structures and Abstractions

Chapter 15

Structs

Every interesting program eventually needs to group related data together. An array of coordinates, a pair of first-and-last names, a record that bundles a user's email with their role—the moment your data grows beyond a single value, you reach for a *struct*.

Lattice structs are lightweight, named containers for fields. They are pass-by-value, they support methods through trait implementations and callable fields, and they integrate naturally with the phase system you met in Chapter 11. In this chapter we will define structs, read and write their fields, attach behavior, customise equality, and peer under the hood with the reflection API.

15.1 Defining and Instantiating Structs

A struct definition gives a name and a list of typed fields:

Listing 15.1: A minimal struct definition

```
struct Point { x: any, y: any }
```

The keyword `struct` is followed by a name that must begin with an uppercase letter, then a pair of braces enclosing comma-separated fields. Each field has a name and a type annotation. Right now the most common annotation is `any`, which accepts every type. Field annotations are documentation—the runtime does not police them—so `any` costs you nothing but clarity. Chapter 18 surveys the wider type syntax, including the parts that are reserved for a future type checker.

Struct

A *struct* is a named, composite value type whose data is stored in a fixed set of named fields. Structs are **pass-by-value**: assigning a struct to a new variable produces an independent copy.

Once defined, you create an instance with a *struct literal*—the struct name followed by braces and field initialisers:

Listing 15.2: Creating struct instances

```
struct Point { x: any, y: any }

let origin = Point { x: 0, y: 0 }
let target = Point { x: 42, y: 17 }

print(origin)  // Point { x: 0, y: 0 }
print(target)  // Point { x: 42, y: 17 }
```

Every field must be supplied when constructing an instance—there are no default values at the language level. If you want defaults, write a factory function:

Listing 15.3: A factory function with defaults

```
struct Config {
    host: String,
    port: Int,
    verbose: Bool
}

fn default_config() -> Config {
    return Config {
        host: "localhost",
        port: 8080,
        verbose: false
    }
}

let cfg = default_config()
print(cfg.port)  // 8080
```

15.1.1 Nested Structs

Fields can hold any value, including other structs:

Listing 15.4: Nested structs

```
struct Point { x: any, y: any }
struct Rect { origin: any, width: any, height: any }

let r = Rect {
    origin: Point { x: 5, y: 10 },
    width: 100,
    height: 50
}

print(r.origin.x)  // 5
print(r.width)     // 100
```

Nesting is a natural way to compose data. Because structs are pass-by-value, the `Point` inside the `Rect` is a full copy, not a reference. Modifying the original `Point` after constructing the `Rect` has no effect on `r.origin`.

15.1.2 Structs with Typed Fields

Although `any` works everywhere, you can annotate fields with concrete types to state the intended shape:

Listing 15.5: Typed struct fields

```
struct Entity {
    id: Int,
    name: String,
    x: Int,
    y: Int,
    hp: Int,
    max_hp: Int,
    tag: String
}
```

This reads like a blueprint: an `Entity` carries an integer ID, a name, a position, health, and a tag string. Readers of your code will thank you for the specificity.

Field Annotations Are Not Enforced

As of v0.6.4, field type annotations are documentation only. Building `Config { host: 42 }` against a `host: String` field succeeds without a murmur. Function signatures are the opposite—there the runtime *does* check, and a struct name works as a type:

```
struct Config { host: String, port: Int }

fn show(c: Config) -> String { return c.host }

print(show(42))
// vm error: function 'show' parameter 'c' expects type Config, got Int
```

So annotate fields for your readers, and annotate parameters for the runtime.

15.2 Field Access and Mutation

Reading a field uses dot notation:

Listing 15.6: Field access

```
struct Point { x: any, y: any }

let p = Point { x: 3, y: 4 }
print(p.x)      // 3
print(p.y)      // 4
print(p.x + p.y) // 7
```

15.2.1 Mutation and the Phase System

Structs follow the phase system. A struct bound with `fix` (or frozen later with `freeze()`) is *crystallised*—writing to its fields is a runtime error. A `let` binding leaves the struct *unphased*, so field writes are permitted, but when we *intend* to mutate, we signal it with `flux`:

Listing 15.7: Mutating struct fields

```
struct Point { x: any, y: any }

flux p = Point { x: 1, y: 2 }
print(p.x)  // 1

p.x = 99
print(p.x)  // 99

p.y = 88
print(p.y)  // 88
```

Attempting to write through a crystal binding fails loudly. Freezing is enforced on every backend as of v0.5.0:

Listing 15.8: Field writes on a crystal struct error

```
struct Point { x: any, y: any }

fix locked = Point { x: 1, y: 2 }
locked.x = 99
// vm error: cannot assign to field 'x' on a frozen struct
```

The prefix tells you which backend spoke: `vm error:` from the default stack VM, `error:` from `--tree-walk`, and `regvm error:` from `--regvm`. This particular refusal is one of the few whose wording still differs between them—the tree-walker says "of frozen struct" and the register VM says "cannot set field `'x'` on a frozen value"—but all three refuse.

Compound assignment operators work on fields too:

Listing 15.9: Compound field assignment

```
struct Point { x: any, y: any }

flux p = Point { x: 10, y: 20 }
p.x += 5
print(p.x)  // 15
p.y -= 10
print(p.y)  // 10
p.x *= 2
print(p.x)  // 30
```

Mutation is Shallow

When you mutate a field on a `flux` struct, you are updating *that copy's* field. Because structs are pass-by-value, any other variable that received a copy of the struct is unaffected. This is one of the most important mental-model points in Lattice. We will return to it in Section 15.8.

Nested Containers Inside Fields

Writing a whole field (`p.x = 99`) always works on a fluid struct. Mutating a container *inside* a field in place is another story: on both bytecode backends, `b.value.push(4)` and `b.value[0] = 9` operate on a copy of the field, and the change is quietly discarded. (The tree-walking interpreter mutates in place here, so the backends genuinely diverge in v0.6.4.) The portable idiom is to pull the container out, change it, and assign it back:

```
struct Inventory { items: any }

flux inv = Inventory { items: ["axe"] }
flux updated = inv.items
updated.push("rope")
inv.items = updated
print(inv.items)  // [axe, rope]
```

Note that this is a *struct-field* limitation, not a general one. Array elements and map values learned to write back in v0.6.4, so `grid[0].push(9)` and `m["k"].push(2)` do mutate the nested container on all three backends. Struct fields have not caught up.

15.2.2 Optional Field Access

If you have a value that might be `nil`, the optional-chaining operator `?.` lets you access fields safely:

Listing 15.10: Optional field access

```
struct User { name: any, email: any }

fn find_user(id: any) -> any {
    if id == 1 {
        return User { name: "Alice", email: "alice@example.com" }
    }
    return nil
}

let user = find_user(42)
print(user?.name)   // nil (no crash)
print(user?.email)  // nil
```

If the receiver is `nil`, the entire expression evaluates to `nil` instead of producing a runtime error. This pairs well with the `??` nil-coalescing operator covered in Chapter 5.

15.2.3 Passing Structs to Functions

Structs can be passed as function arguments and returned from functions. Remember: each pass produces an independent copy.

Listing 15.11: Structs as arguments and return values

```
struct Point { x: any, y: any }

fn sum_point(pt: any) -> any {
    return pt.x + pt.y
}

fn make_point(a: any, b: any) -> Point {
    return Point { x: a, y: b }
}

let p = Point { x: 3, y: 4 }
print(sum_point(p))  // 7

let q = make_point(100, 200)
print(q.x)  // 100
print(q.y)  // 200
```

15.2.4 Structs in Collections

Arrays, maps, and sets can all hold structs:

Listing 15.12: An array of structs

```
struct Point { x: any, y: any }

let points = [
    Point { x: 1, y: 2 },
    Point { x: 3, y: 4 },
    Point { x: 5, y: 6 }
]

for pt in points {
    print(pt.x + pt.y)
}
// Output: 3, 7, 11
print(len(points))  // 3
```

15.3 Methods via Traits: `impl` Blocks

Lattice attaches behaviour to structs through *trait implementations*. A trait declares a set of method signatures; an `impl` block provides the bodies for a specific struct. We will explore traits in depth in Chapter 17, but here is enough to get you started.

Listing 15.13: A trait and its implementation

```
struct Point { x: any, y: any }

trait Describable {
    fn describe(self: any) -> any
}

impl Describable for Point {
    fn describe(self: any) -> any {
        return "Point(" + to_string(self.x) + ", " + to_string(self.y) + ")"
    }
}

let p = Point { x: 7, y: 8 }
print(p.describe())  // Point(7, 8)
```

The first parameter of every method is `self`, which receives a copy of the struct instance the method was called on. Inside the body you read fields with `self.x`, `self.y`, and so on.

Methods Return New Values

Because `self` is a copy, methods that "modify" the struct must return a new instance. The original is untouched:

```
trait Scalable {
    fn scale(self: any, factor: any) -> any
}

impl Scalable for Point {
    fn scale(self: any, factor: any) -> any {
        return Point { x: self.x * factor, y: self.y * factor }
    }
}

let p = Point { x: 5, y: 10 }
let q = p.scale(3)
print(q.x)  // 15
print(q.y)  // 30
print(p.x)  // 5  (unchanged)
```

15.3.1 Multiple Traits per Struct

A single struct can implement as many traits as you like:

Listing 15.14: A struct implementing multiple traits

```
struct Point { x: any, y: any }

trait HasArea {
    fn area(self: any) -> any
}

trait Describable {
    fn describe(self: any) -> any
}

impl HasArea for Point {
    fn area(self: any) -> any {
        return self.x * self.y
    }
}

impl Describable for Point {
    fn describe(self: any) -> any {
        return "Point(" + to_string(self.x) + ", " + to_string(self.y) + ")"
    }
}

let p = Point { x: 3, y: 4 }
print(p.area())      // 12
print(p.describe())  // Point(3, 4)
```

Under the hood, the compiler registers each method with a key of the form `TypeName::method_name` in the global scope. When you call `p.describe()`, the VM looks up `Point::describe` and invokes it with `self` bound to a clone of `p`. You can see how method dispatch is compiled in `src/stackcompiler.c`—look for the `OP_INVOKE` family of opcodes.

15.3.2 Methods in Loops

Methods compose naturally with iteration:

Listing 15.15: Calling methods in a loop

```
struct Point { x: any, y: any }

trait HasArea {
    fn area(self: any) -> any
}

impl HasArea for Point {
    fn area(self: any) -> any {
        return self.x * self.y
    }
}

let points = [
    Point { x: 1, y: 1 },
    Point { x: 2, y: 3 },
    Point { x: 4, y: 5 }
]

flux total = 0
for p in points {
    total = total + p.area()
}
print(total)  // 1 + 6 + 20 = 27
```

15.4 Callable Struct Fields

Besides attaching methods through traits, you can embed closures directly as struct fields. Any field whose value is a closure becomes *callable*: when you invoke it with dot syntax, Lattice finds the closure field and calls it.

Listing 15.16: A struct with a callable closure field

```
struct Greeter {
    greeting: String,
    greet: Fn
}

let hello = Greeter {
    greeting: "Hello",
    greet: |self| {
        self.greeting + ", world!"
    }
}

print(hello.greet())  // Hello, world!
```

The closure receives `self` as its first argument when called through dot notation, just like a trait method. The difference is that the closure is stored *inside* the struct value itself—it travels with the data.

15.4.1 Stateful Callable Fields

Callable fields shine when you want per-instance behaviour. Consider a vending machine where each operation is a closure:

Listing 15.17: Vending machine with callable fields

```
struct VendingMachine {
    balance: Int,
    inventory: Map,
    state: String,
    insert_coin: Fn,
    select_item: Fn,
    dispense: Fn,
    display: Fn
}

fn make_vending_machine() -> VendingMachine {
    flux inv = Map::new()
    inv.set("cola", 150)
    inv.set("chips", 100)
    inv.set("candy", 75)

    return VendingMachine {
        balance: 0,
        inventory: inv,
        state: "idle",
        insert_coin: |self, amount| {
            self.balance + amount
        },
        select_item: |self, item| {
            if !self.inventory.has(item) {
                "error: unknown item"
            } else {
                let price = self.inventory.get(item)
                if self.balance < price {
                    "error: insufficient funds"
                } else {
                    "ok:" + item
                }
            }
        },
        dispense: |self, item| {
            let price = self.inventory.get(item)
            let change = self.balance - price
            "Dispensing " + item + "! Change: " + to_string(change) + " cents"
        },
        display: |self| {
            "Balance: " + to_string(self.balance) + " cents"
        }
    }
}
```

Each closure can read `self.balance` and `self.inventory`. The caller updates the struct's mutable state after each operation:

Listing 15.18: Driving the vending machine

```
flux vm = make_vending_machine()
print(vm.display())  // Balance: 0 cents

vm.balance = vm.insert_coin(25)
vm.balance = vm.insert_coin(25)
vm.balance = vm.insert_coin(25)
print(vm.display())  // Balance: 75 cents

let result = vm.select_item("candy")
print(result)  // ok:candy
print(vm.dispense("candy"))  // Dispensing candy! Change: 0 cents
```

Notice that the caller writes `vm.balance = vm.insert_coin(25)` rather than letting the closure update the balance itself. That is not ceremony—it is the only thing that works:

`self` Is Always a Clone

Whether a method arrives through a trait impl or a callable field, the runtime hands it a deep clone of the receiver. Writes to `self` land on the clone and evaporate when the call returns:

```
struct Counter { n: Int, bump: Fn }

flux c = Counter { n: 0, bump: |self| { self.n = self.n + 1; self.n } }
print(c.bump())  // 1   (the clone was incremented)
print(c.n)       // 0   (the original never moved)
```

All three backends agree. Methods that need to "change" a struct return the new value, and the caller stores it.

Trait Methods vs. Callable Fields

Use **trait methods** when the behaviour is shared across multiple structs that implement the same interface. Use **callable fields** when the behaviour is unique to each *instance*, or when you want to swap behaviour at runtime by assigning a different closure to the field.

15.4.2 How the Runtime Resolves Calls

When you write `vm.display()`, the runtime checks two places:

1. **Callable field**: Does the struct have a field named `display` whose value is a closure? If so, call it, passing the struct as `self`.

2. **Trait method**: Is there an impl method registered as `VendingMachine::display`? If so, call that with a clone of the struct as `self`.

Callable fields take priority. If a struct has both a closure field named `area` and a trait method `area`, the closure field wins. You can inspect how this dispatch works in `src/eval.c`—the method-call handler first scans `field_names` for a matching closure before falling through to the impl registry.

15.5 Custom Equality with `eq`

By default, two structs are equal if they have the same name and all their fields are recursively equal:

Listing 15.19: Default struct equality

```
struct Point { x: any, y: any }

let a = Point { x: 1, y: 2 }
let b = Point { x: 1, y: 2 }
let c = Point { x: 3, y: 4 }

print(a == b)  // true
print(a == c)  // false
print(a != c)  // true
```

This is structural equality—it compares data, not identity. Two distinct instances with the same field values are considered equal.

15.5.1 Overriding Equality with an `eq` Field

Sometimes you want a different notion of equality. Perhaps two users should be equal if they share the same ID, regardless of other fields. Lattice lets you override equality by adding a closure field named `eq`:

Listing 15.20: Custom equality via the eq field

```
struct User {
    id: Int,
    name: String,
    email: String,
    eq: Fn
}

fn make_user(id: Int, name: String, email: String) -> User {
    return User {
        id: id,
        name: name,
        email: email,
        eq: |self, other| {
            self.id == other.id
        }
    }
}

let alice1 = make_user(1, "Alice", "alice@example.com")
let alice2 = make_user(1, "Alice A.", "alice.a@example.com")
let bob = make_user(2, "Bob", "bob@example.com")

print(alice1 == alice2)  // true  (same ID)
print(alice1 == bob)     // false (different ID)
```

The `eq` closure receives two arguments: `self` (the left-hand operand) and `other` (the right-hand operand). Its return value is converted to a boolean. The `!=` operator automatically negates the result.

The eq Field Must Be a Closure

The custom equality mechanism specifically checks for a field named `eq` whose value is a `VAL_CLOSURE`. If you name a non-closure field `eq`, it will not be used for equality comparisons—the default structural equality applies instead.

15.5.2 When Custom Equality Matters

Custom equality is valuable for:

- **Identity-based comparison**: Compare by ID or key rather than every field.
- **Ignoring computed fields**: Skip over cached or derived data that should not affect equality.

- **Tolerance-based comparison**: For structs containing floating-point values, you might compare with a tolerance:

Listing 15.21: Approximate equality for geometry

```
struct Coordinate {
    lat: Float,
    lng: Float,
    eq: Fn
}

fn make_coord(lat: Float, lng: Float) -> Coordinate {
    return Coordinate {
        lat: lat,
        lng: lng,
        eq: |self, other| {
            let dlat = self.lat - other.lat
            let dlng = self.lng - other.lng
            if dlat < 0 { dlat = 0.0 - dlat }
            if dlng < 0 { dlng = 0.0 - dlng }
            dlat < 0.0001 && dlng < 0.0001
        }
    }
}

let a = make_coord(51.5074, -0.1278)
let b = make_coord(51.5074, -0.1278)
print(a == b)  // true
```

15.6 Struct Reflection

Lattice provides four built-in functions for inspecting and transforming structs at runtime. These live in the category of *reflection*—code that examines the shape of data programmatically.

15.6.1 `struct_name()`

Returns the type name of a struct as a string:

Listing 15.22: Getting a struct's type name

```
struct Point { x: any, y: any }
struct Rect { width: any, height: any }

let p = Point { x: 10, y: 20 }
let r = Rect { width: 5, height: 3 }

print(struct_name(p))  // Point
print(struct_name(r))  // Rect
```

This is useful for logging, serialisation, and dynamic dispatch.

15.6.2 `struct_fields()`

Returns an array of field-name strings:

Listing 15.23: Listing a struct's fields

```
struct Config {
    host: String,
    port: Int,
    verbose: Bool
}

let cfg = Config { host: "localhost", port: 8080, verbose: false }
let fields = struct_fields(cfg)
print(fields)  // [host, port, verbose]
```

The order of names matches the order in the struct definition.

15.6.3 `struct_to_map()`

Converts a struct into a `Map` where keys are field names and values are field values:

Listing 15.24: Converting a struct to a map

```
struct Point { x: any, y: any }

let p = Point { x: 42, y: 17 }
let m = struct_to_map(p)

print(m)            // {"y": 17, "x": 42}
print(m.get("x"))   // 42
print(m.get("y"))   // 17
```

Note that the printed key order follows the map's internal layout, not the struct's field order. This is the fastest way to bridge between structs and the map-based world of JSON serialisation (see Chapter 23).

15.6.4 `struct_from_map()`

Goes the other direction—creates a struct from a type name and a map:

Listing 15.25: Creating a struct from a map

```
struct Point { x: any, y: any }

flux m = Map::new()
m.set("x", 100)
m.set("y", 200)

let p = struct_from_map("Point", m)
print(p.x)  // 100
print(p.y)  // 200
```

If a field is missing from the map, it defaults to `nil`. The struct definition must already exist in scope—`struct_from_map` looks up the registered struct metadata at runtime.

15.6.5 A Practical Reflection Example

Combining these functions, you can write a generic "debug print" utility:

Listing 15.26: A generic struct inspector

```
struct Point { x: any, y: any }
struct User { name: any, age: any }

fn inspect(val: any) -> String {
    let name = struct_name(val)
    let fields = struct_fields(val)
    let m = struct_to_map(val)
    flux parts = []
    for f in fields {
        parts.push(f + "=" + to_string(m.get(f)))
    }
    return name + "(" + parts.join(", ") + ")"
}

let p = Point { x: 3, y: 4 }
let u = User { name: "Alice", age: 30 }

print(inspect(p))  // Point(x=3, y=4)
print(inspect(u))  // User(name=Alice, age=30)
```

Reflection at Runtime

Struct reflection functions operate at *runtime*. The compiler stores field metadata as a hidden global (keyed as `__struct_StructName` in the constant pool). This means reflection incurs a small cost, but it works for any struct, even those defined in imported modules. See `src/stackcompiler.c` for how metadata is emitted during compilation.

15.7 Struct Destructuring

When you want to extract multiple fields at once, destructuring is more concise than repeated dot access:

Listing 15.27: Struct destructuring

```
struct Point { x: any, y: any }

let p = Point { x: 55, y: 66 }
let { x, y } = p
print(x)  // 55
print(y)  // 66
```

The `let { x, y } = p` syntax binds local variables `x` and `y` to the values of `p.x` and `p.y` respectively. The variable names must match the field names.

Destructuring works in any binding context where patterns are accepted, including `match` arms (see Chapter 9).

Selective Destructuring

You do not have to destructure every field. Extracting only the fields you need is perfectly valid:

```
struct Rect { origin: any, width: any, height: any }

let r = Rect { origin: Point { x: 0, y: 0 }, width: 100, height: 50 }
let { width, height } = r
print(width * height)  // 5000
```

15.8 Pass-by-Value Semantics

This is the section that will save you from the most common struct-related confusion. Lattice structs are **pass-by-value**. Every assignment, every function argument, every return value produces an *independent copy* of the struct. There are no hidden references, no shared state, no aliasing surprises.

Listing 15.28: Pass-by-value in action

```
struct Point { x: any, y: any }

flux a = Point { x: 1, y: 2 }
flux b = a          // b is a copy of a
b.x = 999

print(a.x)  // 1   (unchanged)
print(b.x)  // 999
```

After `flux b = a`, the variables `a` and `b` hold entirely separate structs. Mutating one has zero effect on the other.

15.8.1 Implications for Function Calls

When you pass a struct to a function, the function receives its own copy:

Listing 15.29: Functions receive copies

```
struct Counter { value: any }

fn try_to_increment(c: any) {
    flux local = c
    local.value = local.value + 1
    print(local.value)  // 11
}

let counter = Counter { value: 10 }
try_to_increment(counter)
print(counter.value)  // 10 (unchanged)
```

The function's local copy is incremented, but the caller's `counter` retains its original value. If you want the change to be visible to the caller, the function must *return* the new struct:

Listing 15.30: Returning the modified struct

```
struct Counter { value: any }

fn increment(c: any) -> Counter {
    return Counter { value: c.value + 1 }
}

flux counter = Counter { value: 10 }
counter = increment(counter)
print(counter.value)  // 11
```

15.8.2 Why Pass-by-Value?

You might wonder why Lattice chose value semantics when many popular languages use references. The answer ties into the phase system:

- **No aliasing bugs**: Two variables never point at the same struct, so freezing one cannot affect another.
- **Thread safety**: In concurrent code (Chapter 19), spawned tasks receive their own copy of any struct, eliminating data races by construction.
- **Predictability**: When you see `let x = y`, you know that `x` is independent. No need to check whether the type is a "value type" or a "reference type."

Value Semantics

In Lattice, structs (and buffers) are **value types**. Assignment copies the entire struct, including all nested data. The runtime uses `value_deep_clone()` in `include/value.h` to produce the copy. For large structs this means more memory, but it guarantees isolation.

15.8.3 Explicit Copies with `clone()`

If you want to be explicit about copying—perhaps for readability in performance-sensitive code—you can call the `clone()` builtin:

Listing 15.31: Explicit cloning

```
struct Point { x: any, y: any }

let original = Point { x: 1, y: 2 }
flux copy = clone(original)
copy.x = 42

print(original.x)  // 1
print(copy.x)      // 42
```

In practice, assignment already copies, so `clone()` is often a documentation signal: "I know this is making a copy, and I intend that." It earns its keep with frozen values: since v0.6.0 crystal structs are shared by reference under the hood, and `clone()` is the tool that detaches a physically independent copy from the shared structure. `clone()` itself preserves phase—the copy of a crystal comes back crystal. What melts it is the `flux` binding on the next line, which privatizes the value and thaws the copy while leaving the original crystal:

Listing 15.32: Detaching a working copy from a crystal

```
struct Point { x: any, y: any }

fix frozen = Point { x: 1, y: 2 }
flux detached = clone(frozen)
detached.x = 42

print(frozen.x)            // 1
print(detached.x)          // 42
print(phase_of(frozen))    // crystal
print(phase_of(detached))  // fluid
```

15.8.4 When Copies Get Expensive

For small structs like `Point`, copying is negligible. For structs with many fields or large nested arrays, the cost can add up. Strategies for managing this include:

- **Freeze what you share**: As of v0.6.0, crystal (frozen) containers are shared by reference under the hood—assigning or passing a frozen struct is a constant-time retain, while the observable semantics stay identical to copying. Freeze once, pass freely.

- **Keep structs lean**: Store IDs or indices into a central collection rather than embedding large data directly.

- **Use `Ref` for shared state**: When you genuinely need shared mutable access, wrap a value in a `Ref` (covered in Chapter 11).

- **Prefer functional updates**: Return new structs from methods rather than mutating fields in place.

15.9 A Larger Example: Entity Component System

Let's bring everything together with a complete example. An Entity Component System (ECS) is a game architecture pattern where entities are bags of data and systems are functions that transform arrays of entities. It is a perfect showcase for struct-centric design.

Listing 15.33: An ECS in Lattice

```
struct Entity {
    id: Int,
    name: String,
    x: Int,
    y: Int,
    dx: Int,
    dy: Int,
    hp: Int,
    max_hp: Int,
    tag: String
}

fn make_entity(id: Int, name: String, tag: String) -> Entity {
    return Entity {
        id: id, name: name,
        x: 0, y: 0, dx: 0, dy: 0,
        hp: 0, max_hp: 0, tag: tag
    }
}

fn render_entity(e: Entity) {
    flux line = "  [" + to_string(e.id) + "] " + e.name
    line += " at (" + to_string(e.x) + "," + to_string(e.y) + ")"
    if e.max_hp > 0 {
        line += " HP:" + to_string(e.hp) + "/" + to_string(e.max_hp)
    }
    print(line)
}

// System: apply velocity to position
fn movement_system(entities: [Entity]) -> [Entity] {
    flux result = []
    for e in entities {
        flux moved = e
        moved.x = e.x + e.dx
        moved.y = e.y + e.dy
        result.push(moved)
    }
    return result
}
```

The movement system iterates over entities, creates updated copies with new positions, and returns the new array. Because structs are pass-by-value, each `flux moved = e` produces an independent copy that we can mutate without affecting the original.

Listing 15.34: Running the ECS simulation

```
fn main() {
    flux player = make_entity(0, "Hero", "player")
    player.x = 0
    player.y = 0
    player.dx = 1
    player.dy = 1
    player.hp = 100
    player.max_hp = 100

    flux goblin = make_entity(1, "Goblin", "enemy")
    goblin.x = 3
    goblin.y = 3
    goblin.dx = -1
    goblin.dy = -1

    flux entities = [player, goblin]

    flux tick = 0
    while tick < 3 {
        print("--- Tick " + to_string(tick) + " ---")
        for e in entities {
            render_entity(e)
        }
        entities = movement_system(entities)
        tick += 1
    }
}
```

This pattern—immutable-by-default data flowing through pure transformation functions—is idiomatic Lattice. The phase system and value semantics make it safe; the struct system makes it readable.

15.10 Exercises

1. **RGB colour struct.** Define a struct `Color` with fields `r`, `g`, and `b` (all integers 0–255). Write a function `mix(a: Color, b: Color) -> Color` that returns the average of each channel. Verify

that mixing `Color { r: 255, g: 0, b: 0 }` with `Color { r: 0, g: 0, b: 255 }` gives `Color { r: 127, g: 0, b: 127 }`.

2. **Bank account.** Create a `BankAccount` struct with fields `owner`, `balance`, and callable fields `deposit` and `withdraw`. The `withdraw` field should return an error string if the balance is insufficient. Drive a sequence of transactions and print the final balance.

3. **Custom equality.** Define a `Student` struct with fields `id`, `name`, and `gpa`. Add an `eq` closure that compares students by `id` only. Demonstrate that two `Student` instances with the same ID but different GPAs are considered equal.

4. **Reflection round-trip.** Write a function that takes any struct, converts it to a map with `struct_to_map`, doubles every numeric value in the map, and converts it back with `struct_from_map`. Test it with `Point { x: 5, y: 10 }` and verify the result is `Point { x: 10, y: 20 }`.

5. **Value semantics drill.** Without running the code, predict the output of this program, then verify your prediction:

```
struct Box { value: any }
flux a = Box { value: [1, 2, 3] }
flux b = a
b.value = [1, 2, 3, 4]
print(len(a.value))
print(len(b.value))
```

What's Next

Structs give your data a name and a shape. But what if a value can be *one of several shapes*? A network response might be a success or a failure. A drawing command might be a circle, a rectangle, or a line. In the next chapter we meet *enums*—Lattice's answer to sum types—and discover how to model these "one of many" scenarios with precision and safety.

Chapter 16

Enums

A struct says "this value has *all* of these fields." An enum says "this value is *one* of these variants." Where structs model *product types*—a user has a name *and* an age *and* an email—enums model *sum types*: a network response is either a success *or* a failure, a shape is a circle *or* a rectangle *or* a triangle.

Lattice enums are full algebraic data types. Each variant can carry its own payload, and the compiler warns when a `match` lacks a catch-all arm to make it total. In this chapter we will define enums, construct variants, destructure them with pattern matching, model state machines, and build the ubiquitous `Option` and `Result` patterns.

16.1 Defining Enums With and Without Payloads

An enum definition lists a type name and a set of *variants*:

Listing 16.1: A basic enum with unit variants

```
enum Direction {
    North,
    South,
    East,
    West
}
```

Each variant is a distinct tag. A `Direction` value is exactly one of `North`, `South`, `East`, or `West`—nothing else.

Enum

An *enum* (short for "enumeration") is a type that can be one of a fixed set of *variants*. Each variant has a name and may carry a *payload*—zero or more associated values that travel with the tag.

Variants without payloads are called *unit variants*. They carry no additional data:

Listing 16.2: Unit variants carry no data

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

let favourite = Color::Red
print(favourite)  // Color::Red
```

16.1.1 Variants with Payloads

A variant becomes more interesting when it carries data. List the payload types in parentheses after the variant name:

Listing 16.3: Enum with payload variants

```
enum Color {
    Red,
    Green,
    Blue,
    Rgb(any, any, any)
}

let coral = Color::Rgb(255, 127, 80)
print(coral)  // Color::Rgb(255, 127, 80)
```

The `Rgb` variant carries three values—the red, green, and blue channels. Notice that unit variants and payload variants can coexist in the same enum. You can mix and match freely.

Here is a `Shape` enum where each variant carries different amounts of data:

Listing 16.4: Variants with different payload sizes

```
enum Shape {
    Circle(any),
    Rectangle(any, any),
    Triangle(any, any, any),
    None
}
```

`Circle` takes one value (the radius), `Rectangle` takes two (width and height), `Triangle` takes three (side lengths), and `None` carries nothing.

Payload Type Annotations

In the examples above we use `any` for payload types. You can also write concrete types:

```
enum Shape {
    Circle(Float),
    Rectangle(Float, Float),
    None
}
```

This makes the intended contract clearer to a reader. It does not make it enforceable: as of v0.6.4 payload annotations are never checked, so `Shape::Circle("not a float")` constructs happily on every backend—strict mode included, since that checker polices phases rather than types. Generic type parameters (e.g., `Option<T>`) are covered in Chapter 18, along with the one position where a type annotation *is* checked.

16.2 Constructing Variants

To create an enum value, use the double-colon syntax: `EnumName::VariantName` for unit variants, or `EnumName::VariantName(args)` for payload variants.

Listing 16.5: Constructing enum variants

```
enum Color {
    Red,
    Green,
    Blue,
    Rgb(any, any, any)
}

let r = Color::Red
let g = Color::Green
let custom = Color::Rgb(255, 128, 0)

print(r)       // Color::Red
print(g)       // Color::Green
print(custom)  // Color::Rgb(255, 128, 0)
```

16.2.1 Enums Returned from Functions

Enums are regular values—you can pass them to functions, return them, and store them in collections:

Listing 16.6: Returning enums from functions

```
enum Color {
    Red,
    Green,
    Blue,
    Rgb(any, any, any)
}

fn make_color(kind: String) -> Color {
    if kind == "red" {
        return Color::Red
    }
    if kind == "custom" {
        return Color::Rgb(1, 2, 3)
    }
    return Color::Blue
}

print(make_color("red"))     // Color::Red
print(make_color("custom"))  // Color::Rgb(1, 2, 3)
print(make_color("other"))   // Color::Blue
```

16.2.2 Enums in Collections

Listing 16.7: An array of enum values

```
enum Status {
    Pending,
    Running(any),
    Done(any),
    Failed(any, any)
}

let tasks = [
    Status::Pending,
    Status::Running("compile"),
    Status::Done(42),
    Status::Failed("timeout", 504)
]

for task in tasks {
    print(task)
}
// Status::Pending
// Status::Running(compile)
// Status::Done(42)
// Status::Failed(timeout, 504)
```

Notice that string payloads print without quotes—`print` renders the payload the same way it renders a bare string.

16.2.3 Runtime Introspection Methods

Every enum value carries metadata that you can inspect at runtime:

Listing 16.8: Enum introspection methods

```
enum Option {
    Some(any),
    None
}

let val = Option::Some(42)

print(val.tag())           // Some
print(val.enum_name())     // Option
print(val.variant_name())  // Some
print(val.payload())       // [42]
print(val.is_variant("Some"))  // true
print(val.is_variant("None"))  // false
```

The `.payload()` method returns an array of the variant's payload elements. For unit variants, it returns an empty array. These methods are useful for logging and debugging, but for most logic you should prefer pattern matching.

16.3 Matching on Enums

Pattern matching is the primary way to work with enums. If you read Chapter 9, you have already seen `match` expressions with literals, wildcards, and guards. Enum patterns add a new dimension: you can match on the variant *and* destructure the payload in a single step.

Since v0.3.29 the payload position accepts the same vocabulary as any other pattern—bindings, wildcards, and literals—so a single `match` can sort a value by variant and by payload shape at once:

Listing 16.9: The full destructuring vocabulary in one match

```
enum Shape { Circle(Float), Rect(Float, Float), Point }

fn area(s: Shape) -> Float {
    match s {
        Shape::Circle(0.0)   => 0.0,          // literal sub-pattern
        Shape::Circle(r)     => 3.14159 * r * r,  // binding
        Shape::Rect(_, h)    => h * 2.0,      // wildcard sub-pattern
        Shape::Point         => 0.0,          // unit variant
        _ => -1.0
    }
}

print(area(Shape::Circle(0.0)))     // 0
print(area(Shape::Circle(2.0)))     // 12.5664
print(area(Shape::Rect(3.0, 4.0)))  // 8
print(area(Shape::Point))           // 0
```

Arms are tried top to bottom, so the literal `Shape::Circle(0.0)` arm gets first refusal before the general `Shape::Circle(r)` arm sees the value. All three backends agree on every line of that output. The rest of this section takes the pieces one at a time.

16.3.1 Matching Unit Variants

At its most basic, matching an enum looks like matching constants:

Listing 16.10: Matching unit variants

```
enum Direction {
    North,
    South,
    East,
    West
}

fn describe(dir: Direction) -> String {
    match dir {
        Direction::North => "heading north",
        Direction::South => "heading south",
        Direction::East  => "heading east",
        Direction::West  => "heading west",
        _ => "lost"
    }
}

print(describe(Direction::North))  // heading north
print(describe(Direction::West))   // heading west
```

The wildcard arm never fires here—all four variants are listed—but the exhaustiveness checker asks for it anyway. We will see why in Section 16.6.

16.3.2 Destructuring Payloads

When a variant carries data, the match pattern can bind payload elements to variables:

Listing 16.11: Destructuring enum payloads

```
enum Option {
    Some(any),
    None
}

let val = Option::Some(42)

let result = match val {
    Option::Some(x) => "got: " + to_string(x),
    Option::None    => "nothing",
    _ => "unknown"
}
print(result)  // got: 42
```

The pattern `Option::Some(x)` does two things at once: it checks that the value is the `Some` variant, and it binds the payload to the variable `x`. Inside the arm's body, `x` holds the value `42`.

16.3.3 Multi-Element Payloads

Variants with multiple payload elements bind multiple variables:

Listing 16.12: Multi-binding destructuring

```
enum Pair {
    Both(any, any),
    Single(any),
    Empty
}

let p = Pair::Both("hello", 99)
let result = match p {
    Pair::Both(a, b)  => to_string(a) + "=" + to_string(b),
    Pair::Single(x)   => "single: " + to_string(x),
    Pair::Empty        => "empty",
    _ => "?"
}
print(result)  // hello=99
```

16.3.4 Using Destructured Values in Computations

The bound variables are full values—you can use them in any expression:

Listing 16.13: Computing with destructured values

```
enum Result {
    Ok(any),
    Err(any)
}

let r = Result::Ok(21)
let doubled = match r {
    Result::Ok(v)  => v * 2,
    Result::Err(e) => -1,
    _ => 0
}
print(doubled)  // 42
```

16.3.5 Block Bodies in Arms

When a single expression is not enough, use a block body:

Listing 16.14: Block bodies in match arms

```
enum Option {
    Some(any),
    None
}

let val = Option::Some(10)
let result = match val {
    Option::Some(n) => {
        let doubled = n * 2
        let tripled = n * 3
        "doubled=" + to_string(doubled) + " tripled=" + to_string(tripled)
    },
    _ => "none"
}
print(result)  // doubled=20 tripled=30
```

16.3.6 Ignoring Payload with Wildcards

If you care about the variant but not the payload, use _ in the payload position:

Listing 16.15: Ignoring payloads with wildcards

```
enum Status {
    Pending,
    Running(any),
    Done(any),
    Failed(any, any)
}

flux pending_count = 0
flux done_count = 0
let statuses = [Status::Pending, Status::Done(42), Status::Running("build")]

for s in statuses {
    match s {
        Status::Pending => { pending_count += 1 },
        Status::Done(_) => { done_count += 1 },
        _ => {}
    }
}
print(pending_count)  // 1
print(done_count)     // 1
```

16.3.7 Nested Match Expressions

You can nest match expressions to drill into layered data:

Listing 16.16: Nested match on enums

```
enum Result {
    Ok(any),
    Err(any)
}

fn classify(val: any) -> String {
    return match val {
        Result::Ok(v) => {
            match v {
                0 => "ok:zero",
                _ => "ok:" + to_string(v)
            }
        },
        Result::Err(e) => "err:" + to_string(e),
        _ => "unknown"
    }
}

print(classify(Result::Ok(0)))       // ok:zero
print(classify(Result::Ok(42)))      // ok:42
print(classify(Result::Err("bad")))  // err:bad
```

Match as an Expression

Remember that `match` is an expression in Lattice—it evaluates to a value. You can assign its result to a variable, pass it to a function, or return it directly. This makes match-based dispatch composable and concise.

16.4 Using Enums for State Machines

Enums are a natural fit for state machines because each variant represents a distinct state, and the payload carries state-specific data. Let's model a task runner that moves through a lifecycle:

Listing 16.17: A task lifecycle as an enum

```
enum TaskState {
    Pending,
    Running(any),
    Done(any),
    Failed(any, any)
}

fn status_label(s: TaskState) -> String {
    match s {
        TaskState::Pending => "pending",
        TaskState::Running(task) =>
            "running: " + to_string(task),
        TaskState::Done(result) =>
            "done: " + to_string(result),
        TaskState::Failed(err, code) =>
            "failed: " + to_string(err) + " (" + to_string(code) + ")",
        _ => "unknown"
    }
}
```

16.4.1 Driving State Transitions

A state machine advances by replacing the current state with a new variant:

Listing 16.18: State transitions

```
enum TaskState {
    Pending,
    Running(any),
    Done(any),
    Failed(any, any)
}

fn advance(state: TaskState, tick: Int) -> TaskState {
    match state {
        TaskState::Pending => TaskState::Running("job_" + to_string(tick)),
        TaskState::Running(task) => {
            if tick > 3 {
                TaskState::Done("completed " + to_string(task))
            } else {
                TaskState::Running(task)
            }
        },
        _ => state
    }
}

flux state = TaskState::Pending
for tick in 0..6 {
    state = advance(state, tick)
    print(to_string(tick) + ": " + match state {
        TaskState::Pending      => "pending",
        TaskState::Running(t)   => "running " + to_string(t),
        TaskState::Done(r)      => "done: " + to_string(r),
        TaskState::Failed(e, c) => "failed",
        _ => "?"
    })
}
// 0: running job_0
// 1: running job_0
// 2: running job_0
// 3: running job_0
// 4: done: completed job_0
// 5: done: completed job_0
```

Because the state is an enum, every transition site is a `match`, and the wildcard arm gives unhandled states a well-defined default (here: stay put). And because each variant carries exactly the data it needs, you never have stale or irrelevant fields cluttering your state object.

16.4.2 A Parser Token Example

Enums excel at modelling token types in a parser, where each token variant carries different data:

Listing 16.19: Tokens as an enum

```
enum Token {
    Number(any),
    Identifier(any),
    Plus,
    Minus,
    Star,
    LeftParen,
    RightParen,
    Eof
}

fn token_label(tok: Token) -> String {
    match tok {
        Token::Number(n)     => "NUM(" + to_string(n) + ")",
        Token::Identifier(s) => "ID(" + to_string(s) + ")",
        Token::Plus          => "+",
        Token::Minus         => "-",
        Token::Star          => "*",
        Token::LeftParen     => "(",
        Token::RightParen    => ")",
        Token::Eof           => "EOF",
        _ => "?"
    }
}

let tokens = [
    Token::Number(42),
    Token::Plus,
    Token::Identifier("x"),
    Token::Eof
]

for tok in tokens {
    print(token_label(tok))
}
// Output: NUM(42), +, ID(x), EOF
```

16.5 Option-Like and Result-Like Patterns

Two patterns appear so frequently across programming languages that they deserve special attention: *optional values* (a value that might be absent) and *fallible operations* (a computation that might fail). Lattice does not have built-in `Option` and `Result` types in the language itself, but you can define them as enums and they work beautifully.

16.5.1 The Option Pattern

Listing 16.20: Defining and using Option

```
enum Option {
    Some(any),
    None
}

fn find_user(id: Int) -> Option {
    if id == 1 {
        return Option::Some("Alice")
    }
    if id == 2 {
        return Option::Some("Bob")
    }
    return Option::None
}

let user = find_user(1)
match user {
    Option::Some(name) => print("Found: " + name),
    Option::None       => print("User not found"),
    _ => print("?")
}
// Output: Found: Alice
```

An `unwrap_or` helper makes optionals easier to use in expressions:

Listing 16.21: An unwrap_or helper

```
enum Option {
    Some(any),
    None
}

fn unwrap_or(opt: Option, default: any) -> any {
    match opt {
        Option::Some(v) => v,
        Option::None    => default,
        _ => default
    }
}

let a = Option::Some(10)
let b = Option::None

print(unwrap_or(a, 0))   // 10
print(unwrap_or(b, -1))  // -1
```

16.5.2 The Result Pattern

Listing 16.22: Defining and using Result

```
enum Result {
    Ok(any),
    Err(any)
}

fn parse_port(input: String) -> Result {
    if !input.is_digit() {
        return Result::Err("not a number: " + input)
    }
    let n = parse_int(input)
    if n < 1 || n > 65535 {
        return Result::Err("port out of range: " + to_string(n))
    }
    return Result::Ok(n)
}

fn report(r: Result) {
    match r {
        Result::Ok(port)  => print("Listening on port " + to_string(port)),
        Result::Err(msg)  => print("Error: " + msg),
        _ => print("?")
    }
}

report(parse_port("8080"))   // Listening on port 8080
report(parse_port("http"))   // Error: not a number: http
report(parse_port("70000"))  // Error: port out of range: 70000
```

The `is_digit()` guard is doing real work here. It is tempting to call `parse_int` first and test the result for `nil`, but that idiom is not portable: the bytecode backends hand back `nil` for unparseable input while `--tree-walk` raises `parse_int: invalid integer` and never reaches your check. Validating the string before parsing it behaves the same everywhere.

16.5.3 Chaining Results

A common pattern is to chain fallible operations—if the first succeeds, feed its value into the second:

Listing 16.23: Chaining two results

```
enum Result {
    Ok(any),
    Err(any)
}

fn chain_results(a: Result, b: Result) -> Result {
    match a {
        Result::Ok(va) => {
            match b {
                Result::Ok(vb) => Result::Ok(va + vb),
                Result::Err(e) => Result::Err(e),
                _ => Result::Err("unknown")
            }
        },
        Result::Err(e) => Result::Err(e),
        _ => Result::Err("unknown")
    }
}

let r = chain_results(Result::Ok(10), Result::Ok(20))
match r {
    Result::Ok(v)  => print("ok: " + to_string(v)),
    Result::Err(e) => print("err: " + to_string(e)),
    _ => print("unknown")
}
// Output: ok: 30
```

Enums vs. try/catch

Lattice also provides `try`/`catch` for error handling (see Chapter 10). The `Result` pattern and exceptions are complementary strategies. Use `Result` when errors are *expected* and part of your data flow; use `try`/`catch` when errors are *exceptional* and you want to bail out of a computation.

16.6 Exhaustiveness Warnings

A `match` that quietly skips a case is a bug waiting for the right input. Before your program runs, Lattice inspects every `match` expression and warns about the ones that might not cover all values.

16.6.1 How It Works

The checker (in `src/match_check.c`) applies three rules:

1. If any arm is an *unguarded* wildcard (`_`) or binding pattern, the match is exhaustive—the catch-all handles everything.

2. If the arms are `true`/`false` literals, the match is exhaustive once both cases appear; a missing case is named in the warning.

3. Every other match—*including a match on enum variants*—draws a warning asking for a wildcard arm.

That third rule may surprise you: as of v0.6.4, listing every variant of an enum does not satisfy the checker. The reason is a mismatch in the source. `src/match_check.c` contains a per-variant coverage pass that would name the exact variant you forgot—but it recognises a variant arm only when the pattern is a `PAT_LITERAL` wrapping an `EXPR_ENUM_VARIANT`, and the parser never builds patterns that way. Every `Direction::North` arm becomes a `PAT_ENUM_VARIANT` (see `include/ast.h`), which that pass does not see, so the check falls through to the generic advice:

Listing 16.24: All variants listed—the checker still wants a wildcard

```
enum Direction { North, South, East, West }
let d = Direction::East
let l = match d {
    Direction::North => "N",
    Direction::South => "S",
    Direction::East  => "E",
    Direction::West  => "W"
}
print(l)  // E
// warning: non-exhaustive match on line 3: consider adding a wildcard `_` arm
```

The program still runs—this is a warning, not an error—but it arrives with noise on stderr. Add an unguarded `_` arm and the warning disappears. This is why every enum match in this chapter ends with one: the wildcard is the idiomatic full stop of a Lattice `match`.

Guards Disqualify a Catch-All

A guarded wildcard or binding (e.g., `x if x > 0 => ...`) does *not* count as a catch-all, because the guard might fail at runtime. A match built solely from guarded arms warns even when the guards happen to cover every case, and a phase-qualified wildcard is excluded for the same reason. To satisfy the checker, the final arm must be a bare, unguarded _ (or binding).

16.6.2 Boolean Exhaustiveness

The checker also handles boolean matches:

Listing 16.25: Boolean exhaustiveness

```
let flag = true
let label = match flag {
    true  => "yes",
    false => "no"
}
// No warning: both cases covered
```

If you omit a case, the compiler warns with `missing `false` case` (or `missing `true` case`)—the one place where the checker can name exactly what you forgot.

16.6.3 Non-Enum Matches

For integer, string, or float scrutinees, the set of possible values is unbounded. The compiler cannot enumerate them, so it requires a wildcard arm:

Listing 16.26: Wildcard required for integer match

```
let score = 85
let grade = match score {
    0..59   => "F",
    60..69  => "D",
    70..79  => "C",
    80..89  => "B",
    90..100 => "A",
    _       => "invalid"
}
print(grade)  // B
```

Without the `_ => "invalid"` arm, the compiler warns: "consider adding a wildcard _ arm."

16.6.4 Make the Wildcard Arm Loud

Because the v0.6.4 checker cannot tell you *which* variant a match forgot—only that a catch-all is missing—the safety burden shifts to the wildcard arm you write. A wildcard that quietly absorbs surprises hides bugs; a wildcard that produces something unmistakable turns them into reports. Prefer a sentinel you will notice in output or logs (`_ => "BUG: unhandled variant"`) over a plausible-looking default, and route genuinely impossible cases through the error-handling strategies of Chapter 10.

The "Add a Variant" Test

When designing an enum, imagine adding a new variant six months from now. The wildcard arms in your existing matches will swallow it without a peep—so decide *at each match site* what the right default is. Where no default makes sense, make the wildcard arm return a loud sentinel. Your future self, staring at a log line that says `BUG: unhandled variant`, will know exactly where to look.

16.7 Recursive Enums

Because payload elements can be any value, enums can refer to themselves recursively. This is perfect for tree structures:

Listing 16.27: A recursive binary tree

```
enum Tree {
    Leaf(any),
    Branch(any, any)
}

fn tree_sum(t: any) -> Int {
    match t {
        Tree::Leaf(v) => v,
        Tree::Branch(l, r) => tree_sum(l) + tree_sum(r),
        _ => 0
    }
}

fn tree_depth(t: any) -> Int {
    match t {
        Tree::Leaf(_) => 1,
        Tree::Branch(l, r) => {
            let ld = tree_depth(l)
            let rd = tree_depth(r)
            if ld > rd { 1 + ld } else { 1 + rd }
        },
        _ => 0
    }
}

fn make_balanced(depth: Int, value: Int) -> Tree {
    if depth <= 0 {
        return Tree::Leaf(value)
    }
    return Tree::Branch(
        make_balanced(depth - 1, value * 2),
        make_balanced(depth - 1, value * 2 + 1)
    )
}

let tree = make_balanced(3, 1)
print("sum: " + to_string(tree_sum(tree)))    // sum: 92
print("depth: " + to_string(tree_depth(tree)))  // depth: 4
```

Each `Branch` holds two children that are themselves `Tree` values. The recursive functions use match to distinguish leaves from branches, and the wildcard arm handles unexpected cases gracefully.

16.7.1 An Expression Evaluator

Recursive enums can model abstract syntax trees for a mini-language:

Listing 16.28: An expression tree evaluator

```
enum Expr {
    Num(any),
    Add(any, any),
    Mul(any, any),
    Neg(any)
}

fn eval_expr(e: Expr) -> Int {
    match e {
        Expr::Num(n)    => n,
        Expr::Add(a, b) => eval_expr(a) + eval_expr(b),
        Expr::Mul(a, b) => eval_expr(a) * eval_expr(b),
        Expr::Neg(x)    => -eval_expr(x),
        _ => 0
    }
}

// Represents: (3 * 4) + (-(2))
let expr = Expr::Add(
    Expr::Mul(Expr::Num(3), Expr::Num(4)),
    Expr::Neg(Expr::Num(2))
)
print(eval_expr(expr))  // 10
```

This is a pattern you will encounter throughout compiler design, symbolic mathematics, and rule engines. Enums make the variant structure explicit; match makes the dispatch exhaustive.

16.8 Practical Patterns

16.8.1 Enums and Loops

Building enum values inside loops and processing them with match is a common workflow:

Listing 16.29: Batch-constructing enums

```
enum Status {
    Pending,
    Running(any),
    Done(any),
    Failed(any, any)
}

flux statuses = []
for i in 0..8 {
    let s = match i % 4 {
        0 => Status::Pending,
        1 => Status::Running("task_" + to_string(i)),
        2 => Status::Done(i * 10),
        _ => Status::Failed("err_" + to_string(i), i)
    }
    statuses.push(s)
}

flux pending_count = 0
flux done_count = 0
for s in statuses {
    match s {
        Status::Pending => { pending_count += 1 },
        Status::Done(_) => { done_count += 1 },
        _ => {}
    }
}
print("pending: " + to_string(pending_count))  // pending: 2
print("done: " + to_string(done_count))        // done: 2
```

16.8.2 Enums as Function Return Types

Use enums to signal outcomes without exceptions:

Listing 16.30: Validation with enums

```
enum ValidationResult {
    Valid,
    TooShort(any),
    TooLong(any),
    InvalidChar(any)
}

fn validate_username(name: String) -> ValidationResult {
    if len(name) < 3 {
        return ValidationResult::TooShort(len(name))
    }
    if len(name) > 20 {
        return ValidationResult::TooLong(len(name))
    }
    return ValidationResult::Valid
}

let result = validate_username("Al")
match result {
    ValidationResult::Valid         => print("Username accepted"),
    ValidationResult::TooShort(n)  => print("Too short: " + to_string(n) + " chars"),
    ValidationResult::TooLong(n)   => print("Too long: " + to_string(n) + " chars"),
    ValidationResult::InvalidChar(c) => print("Bad char: " + to_string(c)),
    _ => print("?")
}
// Output: Too short: 2 chars
```

16.8.3 Under the Hood

At runtime, an enum value is represented as a tagged union (see `include/value.h`):

- **`enum_name`**: the enum's type name (e.g., "Option")
- **`variant_name`**: the variant (e.g., "Some")
- **`payload`**: an array of values (empty for unit variants)
- **`payload_count`**: the number of payload elements

The compiler emits `OP_BUILD_ENUM` with the enum name, variant name, and payload count as operands. The VM pops payload values from the stack and assembles the enum value. You can explore the compilation logic in `src/stackcompiler.c`—look for the `EXPR_ENUM_VARIANT` case.

16.9 Exercises

1. **Traffic light.** Define an enum `Light` with variants `Red`, `Yellow`, and `Green`. Write a function `next_light(l: Light) -> Light` that cycles through the states: Green → Yellow → Red → Green. Run the cycle 6 times and print each state.

2. **Shape areas.** Define an enum `Shape` with variants `Circle(any)`, `Rectangle(any, any)`, and `Square(any)`. Write a function that returns the area. Use 3.14159 for pi.

3. **Linked list.** Model a linked list with an enum: `enum List { Cons(any, any), Nil }`. Write functions `list_sum` (sum all elements) and `list_len` (count elements). Build a list of `[1, 2, 3, 4, 5]` and verify both functions.

4. **Result pipeline.** Write a function `map_result(r: Result, f: Fn) -> Result` that applies `f` to the value inside `Result::Ok` and passes `Result::Err` through unchanged. Chain three `map_result` calls to build a pipeline.

5. **Exhaustiveness experiment.** Define an enum with five variants. Write a match expression that lists all five variants but has no wildcard arm, run it, and observe the warning. Then experiment with the checker: which final arms silence it? Try an unguarded _, a guarded _ (with `if true`), and a plain binding like `other`.

What's Next

We have seen how to group fields into structs and model alternatives with enums. But so far, attaching methods to a struct has required declaring a trait—even when the trait is implemented by only one type. In the next chapter we explore *traits and impl blocks* in depth: how to declare interfaces, implement them for multiple types, and use trait-driven polymorphism to write code that works across any struct that speaks the right protocol.

Chapter 17

Traits and Impl Blocks

In Chapter 15 we attached behaviour to structs through traits and impl blocks, but we treated them as a recipe to follow without explaining the ingredients. This chapter puts traits centre stage. We will see how to declare interfaces, implement them for multiple types, compose impl blocks, and use traits as the backbone of polymorphic design.

Think of a trait as a *contract*—a promise that a type will provide certain methods. An impl block is the *fulfilment* of that promise. By separating the "what" (the trait) from the "how" (the impl), Lattice lets you write code that operates on *any type that speaks a given protocol*, without knowing the concrete type in advance.

17.1 Declaring Traits

A trait declaration lists method signatures with no bodies:

Listing 17.1: A trait with a single method

```
trait Describable {
    fn describe(self: any) -> String
}
```

The keyword `trait` is followed by a name (conventionally an adjective or capability noun), then braces containing one or more method signatures. Each signature specifies:

- The method name.

- A parameter list. The first parameter is almost always `self: any`, representing the value the method is called on.
- An optional return type.

Trait

A *trait* is a named set of method signatures that defines a behavioural interface. It contains no implementation—only declarations of the methods a type must provide.

17.1.1 Multi-Method Traits

A trait can declare any number of methods:

Listing 17.2: A trait with multiple methods

```
trait Collection {
    fn push_item(self: any, item: any) -> any
    fn pop_item(self: any) -> any
    fn peek_item(self: any) -> any
    fn is_empty(self: any) -> any
}
```

This `Collection` trait declares four methods. Any type that implements `Collection` is expected to provide all four.

17.1.2 Traits with Extra Parameters

Methods are not limited to `self`. They can accept additional parameters:

Listing 17.3: Traits with extra parameters

```
trait Scalable {
    fn scale(self: any, factor: any) -> any
}

trait Combinable {
    fn combine(self: any, other: any) -> any
}
```

The `Scalable` trait expects a method that takes a scaling factor; `Combinable` expects a method that merges two values of the same kind.

Trait Bodies Are Signatures Only

You cannot write method bodies inside a trait declaration. The trait is purely a contract. All implementation goes in `impl` blocks. This is a deliberate design choice: it keeps the interface description separate from the implementation, making both easier to read.

17.2 Implementing Traits for Structs

An `impl` block provides method bodies for a specific type:

Listing 17.4: Implementing a trait

```
struct Animal { kind: any, name: any, legs: any }

trait Displayable {
    fn display(self: any) -> String
}

impl Displayable for Animal {
    fn display(self: any) -> String {
        return self.name + " the " + self.kind
            + " (" + to_string(self.legs) + " legs)"
    }
}

let dog = Animal { kind: "dog", name: "Rex", legs: 4 }
print(dog.display())  // Rex the dog (4 legs)
```

The syntax is `impl TraitName for TypeName { ... }`, and inside the braces you write full function definitions. Each method must match a signature declared in the trait.

17.2.1 How `self` Works

The first parameter, `self`, receives a *copy* of the value the method was called on. This follows Lattice's value semantics:

Listing 17.5: Self is a copy

```
struct Counter { value: any }

trait Incrementable {
    fn increment(self: any) -> Counter
}

impl Incrementable for Counter {
    fn increment(self: any) -> Counter {
        return Counter { value: self.value + 1 }
    }
}

let c = Counter { value: 10 }
let c2 = c.increment()
print(c.value)   // 10 (unchanged)
print(c2.value)  // 11
```

Because `self` is a copy, "mutating" methods must return a new instance. The original remains untouched. This is the same pattern we saw with struct methods in Section 15.3.

17.2.2 Accessing Fields through `self`

Inside a method body, `self` is a regular struct value. You read its fields with dot notation:

Listing 17.6: Reading fields from self

```
struct Rect { w: any, h: any }

trait Geometry {
    fn area(self: any) -> any
    fn perimeter(self: any) -> any
}

impl Geometry for Rect {
    fn area(self: any) -> any {
        return self.w * self.h
    }

    fn perimeter(self: any) -> any {
        return 2 * (self.w + self.h)
    }
}

let r = Rect { w: 5, h: 3 }
print(r.area())       // 15
print(r.perimeter())  // 16
```

17.2.3 Method Chaining

When a method returns a struct of the same type, you can chain calls:

Listing 17.7: Method chaining

```
struct Stack { items: any, size: any }

trait Collection {
    fn push_item(self: any, item: any) -> any
    fn peek_item(self: any) -> any
}

impl Collection for Stack {
    fn push_item(self: any, item: any) -> any {
        flux new_items = []
        for i in 0..self.size {
            new_items.push(self.items[i])
        }
        new_items.push(item)
        return Stack { items: new_items, size: self.size + 1 }
    }

    fn peek_item(self: any) -> any {
        if self.size == 0 {
            return nil
        }
        return self.items[self.size - 1]
    }
}

let s = Stack { items: [], size: 0 }
let top = s.push_item(1).push_item(2).push_item(3).peek_item()
print(top)  // 3
```

Each `push_item` returns a new `Stack`, so we can immediately call the next method on the result. The chain reads like a pipeline: push 1, push 2, push 3, peek.

Chaining Reads Naturally

Method chaining works because Lattice methods that return the same type create a "fluent interface." This is a powerful pattern for building up data structures or transformation pipelines without intermediate variables.

17.3 Multiple Impl Blocks per Struct

A struct can implement as many traits as it needs. Each trait gets its own `impl` block:

Listing 17.8: Three traits on one struct

```
struct Point { x: any, y: any }

trait Describable {
    fn describe(self: any) -> String
}

trait HasArea {
    fn area(self: any) -> any
}

trait Scalable {
    fn scale(self: any, factor: any) -> any
}

impl Describable for Point {
    fn describe(self: any) -> String {
        return "Point(" + to_string(self.x) + ", " + to_string(self.y) + ")"
    }
}

impl HasArea for Point {
    fn area(self: any) -> any {
        return self.x * self.y
    }
}

impl Scalable for Point {
    fn scale(self: any, factor: any) -> any {
        return Point { x: self.x * factor, y: self.y * factor }
    }
}

let p = Point { x: 3, y: 4 }
print(p.describe())  // Point(3, 4)
print(p.area())      // 12

let q = p.scale(3)
print(q.x)  // 9
print(q.y)  // 12
```

The `Point` struct now speaks three protocols: `Describable`, `HasArea`, and `Scalable`. Each impl block is self-contained and can be placed anywhere in the same file (or module).

17.3.1 How Method Registration Works

Under the hood, the compiler registers each method as a global function with a composite key. When you write:

```
impl HasArea for Rect {
    fn area(self: any) -> any { ... }
}
```

the compiler emits the method body as a function and stores it under the key `Rect::area`. When you later call `r.area()`, the VM:

1. Checks for a callable field named `area` on the struct (there is none).
2. Looks up the global `Rect::area`.
3. Calls it with a clone of `r` as `self`.

You can see this in `src/stackcompiler.c`: the `ITEM_IMPL` case builds the key string `"TypeName::methodName"` and emits `OP_DEFINE_GLOBAL` with that key. The VM's `OP_INVOKE` handler resolves the method at call time.

Callable Fields Take Priority

If a struct has both a closure field and a trait method with the same name, the closure field wins. The runtime checks fields first, then falls through to the impl registry. This is by design: it lets you override a trait method on a per-instance basis if needed (see Section 15.4).

17.4 Designing with Interfaces

Traits are Lattice's interface mechanism. Effective use of traits leads to code that is modular, testable, and extensible. Here are some patterns.

17.4.1 The Same Trait on Different Types

The real power of traits emerges when multiple types implement the same interface. Functions that operate on the trait—rather than a specific struct—work with *any* implementing type:

Listing 17.9: One trait, two types

```
struct Circle { radius: Float }
struct Rect { width: Float, height: Float }

trait Describable {
    fn describe(self: any) -> String
}

trait HasArea {
    fn area(self: any) -> Float
}

impl Describable for Circle {
    fn describe(self: any) -> String {
        return "Circle(r=" + to_string(self.radius) + ")"
    }
}

impl HasArea for Circle {
    fn area(self: any) -> Float {
        return 3.14159 * self.radius * self.radius
    }
}

impl Describable for Rect {
    fn describe(self: any) -> String {
        return "Rect(" + to_string(self.width) + "x" + to_string(self.height) + ")"
    }
}

impl HasArea for Rect {
    fn area(self: any) -> Float {
        return self.width * self.height
    }
}

let c = Circle { radius: 5.0 }
let r = Rect { width: 4.0, height: 6.0 }

print(c.describe())  // Circle(r=5)
print(c.area())      // 78.5397

print(r.describe())  // Rect(4x6)
print(r.area())      // 24
```

Both `Circle` and `Rect` implement `Describable` and `HasArea`. The methods have different bodies—each type knows how to describe and measure itself—but the *interface* is identical.

17.4.2 Traits for Separation of Concerns

Suppose you are building a notification system. You might define separate traits for each concern:

Listing 17.10: Separating concerns with traits

```
struct EmailAlert {
    to: String,
    subject: String,
    body: String
}

trait Sendable {
    fn send(self: any) -> String
}

trait Loggable {
    fn log_entry(self: any) -> String
}

impl Sendable for EmailAlert {
    fn send(self: any) -> String {
        return "Sending email to " + self.to + ": " + self.subject
    }
}

impl Loggable for EmailAlert {
    fn log_entry(self: any) -> String {
        return "[EMAIL] to=" + self.to + " subject=" + self.subject
    }
}

let alert = EmailAlert {
    to: "ops@example.com",
    subject: "Server down",
    body: "The production server is unreachable."
}

print(alert.send())      // Sending email to ops@example.com: Server down
print(alert.log_entry()) // [EMAIL] to=ops@example.com subject=Server down
```

Each trait captures one aspect of behaviour. A `SlackAlert` struct could implement the same traits with different bodies, and any code that calls `.send()` or `.log_entry()` would work with either type.

17.4.3 Naming Conventions

Good trait names describe a capability, not a thing:

- `Describable`, `Displayable` — "can be described / displayed"
- `HasArea`, `HasLength` — "possesses a measurable property"
- `Scalable`, `Combinable` — "can be scaled / combined"
- `Collection`, `Geometry` — group of related operations

When you read `impl Scalable for Vec2`, the intent is immediately clear: "`Vec2` can be scaled."

17.5 Trait-Driven Polymorphism

Because Lattice is dynamically typed, any value can be passed to any function. Traits add a *semantic* layer on top: they document which methods a value must support, and the runtime dispatches to the correct implementation based on the value's type.

17.5.1 Heterogeneous Collections

You can put different types in the same array and call the same method on each:

Listing 17.11: Polymorphic dispatch over an array

```
struct Circle { radius: Float }
struct Rect { width: Float, height: Float }

trait HasArea {
    fn area(self: any) -> Float
}

impl HasArea for Circle {
    fn area(self: any) -> Float {
        return 3.14159 * self.radius * self.radius
    }
}

impl HasArea for Rect {
    fn area(self: any) -> Float {
        return self.width * self.height
    }
}

let shapes = [
    Circle { radius: 3.0 },
    Rect { width: 4.0, height: 5.0 },
    Circle { radius: 1.0 }
]

flux total_area = 0.0
for shape in shapes {
    total_area = total_area + shape.area()
}
print(total_area)  // 28.27431 + 20.0 + 3.14159 = 51.4159
```

The `for` loop does not know whether each element is a `Circle` or a `Rect`. It calls `.area()` on each, and the runtime dispatches to the correct implementation based on the struct's type name.

Dynamic Dispatch

Lattice uses *dynamic dispatch* for trait methods. When you call `shape.area()`, the VM looks up the method key `TypeName::area` at runtime, using the actual type of the value. This is different from static dispatch (resolved at compile time) used by languages like Rust.

17.5.2 Methods in Expressions and Conditionals

Trait method calls are expressions—you can use them in `if` conditions, arithmetic, and anywhere else a value is expected:

Listing 17.12: Methods in expressions

```
struct Rect { w: any, h: any }

trait Geometry {
    fn area(self: any) -> any
}

impl Geometry for Rect {
    fn area(self: any) -> any {
        return self.w * self.h
    }
}

let r1 = Rect { w: 4, h: 6 }
let r2 = Rect { w: 3, h: 8 }

let total_area = r1.area() + r2.area()
print(total_area)  // 48

let larger = if r1.area() > r2.area() { "r1" } else { "r2" }
print(larger)  // r2
```

17.5.3 Methods in Loops

A common pattern is to accumulate results by calling trait methods inside a loop:

Listing 17.13: Accumulating over trait methods

```
struct Rect { w: any, h: any }

trait Geometry {
    fn area(self: any) -> any
}

impl Geometry for Rect {
    fn area(self: any) -> any {
        return self.w * self.h
    }
}

let rects = [
    Rect { w: 1, h: 1 },
    Rect { w: 2, h: 2 },
    Rect { w: 3, h: 3 }
]

flux total = 0
for r in rects {
    total = total + r.area()
}
print(total)  // 1 + 4 + 9 = 14
```

17.5.4 Traits on Enums

Traits are not limited to structs—you can implement them for enums too:

Listing 17.14: A trait implemented for an enum

```
enum Status {
    Active,
    Inactive,
    Error(any)
}

trait Displayable {
    fn display(self: any) -> String
}

impl Displayable for Status {
    fn display(self: any) -> String {
        return match self {
            Status::Active     => "active",
            Status::Inactive   => "inactive",
            Status::Error(msg) => "error: " + to_string(msg),
            _ => "unknown"
        }
    }
}

print(Status::Active.display())            // active
print(Status::Error("disk full").display()) // error: disk full
```

Enum Method Dispatch

When you call a trait method on an enum value, the runtime looks up the method key using the *enum type name*—not the variant name. So `Status::Active`, `Status::Inactive`, and `Status::Error("oops")` all dispatch to the same impl block. Per-variant behaviour therefore lives *inside* the method body, as a `match` on `self`—which is exactly what the example above does. There is no way to write one impl per variant.

17.5.5 A Complete Polymorphism Example

Let's put it all together with a two-dimensional vector type that supports scaling and combining:

Listing 17.15: A complete trait-driven design

```
struct Vec2 { x: any, y: any }

trait Describable {
    fn describe(self: any) -> String
}

trait Scalable {
    fn scale(self: any, factor: any) -> any
}

trait Combinable {
    fn combine(self: any, other: any) -> any
}

impl Describable for Vec2 {
    fn describe(self: any) -> String {
        return "Vec2(" + to_string(self.x) + ", " + to_string(self.y) + ")"
    }
}

impl Scalable for Vec2 {
    fn scale(self: any, factor: any) -> any {
        return Vec2 { x: self.x * factor, y: self.y * factor }
    }
}

impl Combinable for Vec2 {
    fn combine(self: any, other: any) -> any {
        return Vec2 { x: self.x + other.x, y: self.y + other.y }
    }
}

let velocity = Vec2 { x: 3, y: 4 }
let boost = velocity.scale(2)
print(boost.describe())  // Vec2(6, 8)

let gravity = Vec2 { x: 0, y: -1 }
let result = boost.combine(gravity)
print(result.describe())  // Vec2(6, 7)
```

This design is extensible. If you later add a `Vec3` struct, you can implement the same three traits for it, and any code that operates on `Describable`, `Scalable`, or `Combinable` values will work with both types—no changes needed.

17.6 Under the Hood

Understanding the implementation helps when debugging method dispatch or designing complex trait hierarchies.

17.6.1 Compilation

Traits themselves are metadata-only. The compiler sees a trait declaration and records it, but emits no bytecode for it. Only `impl` blocks produce code:

1. Each method body is compiled as a regular function.
2. The compiled function is registered as a global under the key `"TypeName::methodName"`.
3. The VM resolves this key at call time via the `OP_INVOKE` opcode family.

You can see this in `src/stackcompiler.c`—the `ITEM_TRAIT` case is essentially a no-op, while `ITEM_IMPL` compiles each method and emits `OP_DEFINE_GLOBAL`.

17.6.2 Dispatch Performance

The stack VM uses three specialised invoke opcodes for performance:

- **`OP_INVOKE_LOCAL`**: When the receiver is a local variable, the VM can skip the stack and access the value directly from the frame's slot array. This is the fastest path.
- **`OP_INVOKE_GLOBAL`**: When the receiver is a global variable. Slightly slower due to the global lookup.
- **`OP_INVOKE`**: The general case for arbitrary expressions. The struct is on the stack, and the VM searches for the method.

Each path also maintains a *polymorphic inline cache* (PIC) to avoid repeated lookups when the same call site sees the same type across iterations. You can explore this in `include/inline_cache.h` and its use in `src/stackvm.c`.

17.6.3 Self Cloning

When a method is invoked, the runtime clones `self` via `value_deep_clone()` before passing it to the method frame. This ensures that mutations inside the method cannot accidentally affect the caller's copy—maintaining Lattice's value semantics guarantee.

17.7 Exercises

1. **Shape hierarchy.** Define structs `Circle`, `Square`, and `Triangle`. Create a `HasArea` trait and implement it for all three. Put instances of each into an array and compute the total area in a loop.

2. **Stack from scratch.** Using the `Collection` trait from this chapter (with methods `push_item`, `pop_item`, `peek_item`, and `is_empty`), implement a `Queue` struct that supports FIFO ordering. Verify that items come out in insertion order.

3. **Composing traits.** Define two traits: `Serializable` (with a `to_string_repr` method) and `Parseable` (with a class-level `from_string` function). Implement both for a `Temperature` struct that holds a value and a unit (`"C"` or `"F"`). Demonstrate round-tripping: convert to a string and parse back.

4. **Trait method chaining.** Create a `Builder` struct with fields `parts` and `count`. Implement a `Buildable` trait with methods `add_part` (returns a new Builder with the part appended) and `build` (returns a summary string). Chain at least four `add_part` calls followed by `build`.

What's Next

Traits let you write code that works across multiple types. But so far, every type annotation in our examples has been `any`—a catch-all that says nothing about what the value actually is. In the next chapter we meet *generics*: the angle-bracket syntax that lets you write `struct Box<T>` and `fn pick<T>(...)`. It is a shorter chapter than you might expect, because in v0.6.4 that syntax is a reservation rather than a feature—the parser understands it, nothing else does, and there are places where using it will stop your program cold. We will map exactly where the line falls.

Chapter 18

Generics: The Syntax Reserved for the Future

Up to this point, every type annotation in our structs, enums, and functions has been either a concrete type like `Int` or the catch-all `any`. Concrete types are precise but inflexible; `any` is flexible but tells the reader nothing about intent. Generics are supposed to occupy the middle ground: code *parameterised* over types, so the shape of the data is documented even when the specific type is left open.

Lattice reserves the syntax for that middle ground. The parser understands `fn pick<T>(...)`, `struct Box<T>`, `enum Option<T>`, and type arguments like `Map<String, Int>`. What it does *not* have, as of v0.6.4, is any machinery behind them. There is no type substitution, no inference, no checking of `T` against anything. A type parameter is a name the parser records and the rest of the compiler ignores.

That would be harmless if the story ended there. It does not. Lattice *does* type-check function parameters and return values at runtime, and that checker has never heard of type parameters—so writing `fn identity<T>(x: T) -> T` produces a function that fails on every call. This chapter maps the boundary precisely: where the generic syntax is free, where it is decorative, and where it will stop your program.

Read This Chapter Before You Use Angle Brackets

Generics in v0.6.4 are not "unenforced but harmless." Used in one specific position—a bare type-parameter name as a function parameter or return type—they turn a working function into one that raises a runtime error the first time you call it. The rule and its boundaries are in Section 18.1.

18.1 Generic Functions and the Runtime Type Checker

Let's start where the trouble is. Here is the canonical generic function, written exactly as you would write it in Rust or TypeScript:

Listing 18.1: The generic identity function—which does not run

```
fn pick<T>(x: T) -> T {
    return x
}

print(pick(42))
// vm error: function 'pick' parameter 'x' expects type T, got Int
//           (did you mean 'Fn'?)
```

The program parses cleanly. It compiles. And then it dies on the first call, on all three backends. (The prefix differs—`vm error:` on the default stack VM, `error:` on `--tree-walk`, `regvm error:` on `--regvm`—but the refusal does not.)

18.1.1 Why This Happens

Lattice checks function parameter and return types *at call time*. You can see the loop in `src/eval.c`, just after the arity and phase checks: for every declared parameter with a type name, the runtime calls `type_matches_value()` and errors if the value does not fit.

That check is a good thing. It is what catches this:

Listing 18.2: Runtime type checking, working as intended

```
struct Config { host: String, port: Int }

fn show(c: Config) -> String { return c.host }

print(show(42))
// vm error: function 'show' parameter 'c' expects type Config, got Int
```

But nothing tells that loop that `T` is a type *parameter* rather than a type *name*. It looks up `T`, finds no such type, decides the value does not match, and—because `T` is not a known built-in—helpfully suggests the nearest name it does know. Hence the slightly surreal "did you mean `'Fn'`?" on a one-letter type parameter.

Type Parameter

A *type parameter* is a placeholder name (conventionally a single uppercase letter like `T`, `U`, or `K`) that stands for a type to be specified later. It appears between angle brackets in a declaration: `fn foo<T>(x: T) -> T`. In v0.6.4 Lattice parses type parameters and stores them in the AST; no later stage substitutes anything for them.

18.1.2 Exactly Where the Line Falls

The runtime checker only inspects two things: the type of each argument against its parameter annotation, and the type of the return value against the return annotation. Everything it inspects, it inspects by *name*. So a type parameter is fatal in precisely the positions the checker reads by name, and invisible everywhere else.

Fatal—a bare type parameter as a parameter or return type:

Listing 18.3: Positions that break every call

```
fn a<T>(x: T) -> any { ... }        // parameter 'x' expects type T
fn b<T>(x: any) -> T { ... }        // return type expects T
fn c<K, V>(key: K, value: V) { ... } // parameter 'key' expects type K
```

Safe—every position the checker does not read as a bare name:

Listing 18.4: Positions where a type parameter is harmless

```
fn head<T>(items: [T]) -> any { return items[0] }

fn dup<T>(value: any, count: Int) -> [T] {
    flux result = []
    for i in 0..count { result.push(value) }
    return result
}

fn transform<T, U>(items: [T], f: Fn(T) -> U) -> [U] {
    flux result = []
    for item in items { result.push(f(item)) }
    return result
}

print(head([10, 20, 30]))                              // 10
print(dup("ha", 3))                                    // [ha, ha, ha]
print(transform(["alice", "bob"], |s| { return len(s) }))  // [5, 3]
```

Inside `[T]`, the checker verifies only that the value is an array—it never descends to the element type. Inside `Fn(T) -> U`, it verifies only that the value is callable. Struct fields and enum payloads are not checked at all. Those are the safe harbours, and they are where the rest of this chapter lives.

The Practical Rule

Declare type parameters freely. Use them in `[T]` element positions, inside `Fn(...)` signatures, on struct fields, and on enum payloads. Never write a bare `T` as a parameter type or a return type—write `any` there and let the angle brackets carry the documentation.

Applying that rule to the identity function gives us something that both runs and reads:

Listing 18.5: A generic signature that survives contact with the runtime

```
fn tag_all<T>(items: [T], label: String) -> [String] {
    flux out = []
    for it in items {
        out.push(label + ":" + to_string(it))
    }
    return out
}

print(tag_all([1, 2, 3], "n"))    // [n:1, n:2, n:3]
print(tag_all(["a", "b"], "s"))   // [s:a, s:b]
```

The `<T>` tells a reader that the element type is deliberately open. The `[T]` annotation says the argument is an array of that element type. Neither costs anything at runtime, and neither is checked.

18.2 Generic Structs

Structs are a safe harbour. Field annotations are never checked at runtime, so a type parameter on a field is pure documentation and behaves perfectly:

Listing 18.6: A generic Box struct

```
struct Box<T> {
    value: T
}

let int_box = Box { value: 42 }
let str_box = Box { value: "hello" }

print(int_box.value)  // 42
print(str_box.value)  // hello
```

The `<T>` after the struct name introduces a type parameter. The field `value` is annotated with `T`, signalling that a `Box` holds exactly one value of a uniform type—a promise to your reader that the language will not check for you.

Instantiation Takes No Type Arguments

When you write `Box { value: 42 }`, you cannot specify `Box<Int> { value: 42 }`—that is a parse error (`unexpected token ':' in expression`). The type parameter exists in the *definition* only; at construction time, the value itself determines what is actually stored.

18.2.1 Multi-Parameter Structs

Listing 18.7: A struct with two type parameters

```
struct Pair<A, B> {
    first: A,
    second: B
}

let entry = Pair { first: "name", second: 42 }
print(entry.first)   // name
print(entry.second)  // 42
```

The `Pair<A, B>` struct makes it clear that the two fields may hold different types. Contrast this with:

```
struct Pair { first: any, second: any }
```

Both definitions behave identically at runtime, but the generic version tells the reader "these are distinct type roles, not arbitrary values."

18.2.2 Realistic Generic Structs

Let's build a generic key–value entry. Note what the factory function does—and does not—annotate:

Listing 18.8: Generic key-value entry

```
struct Entry<K, V> {
    key: K,
    value: V
}

// The <K, V> documents the roles; the parameters stay `any`, because a
// bare K or V here would fail the runtime type check on every call.
fn make_entry<K, V>(key: any, value: any) -> Entry {
    return Entry { key: key, value: value }
}

let user_entry = make_entry("alice", 30)
print(user_entry.key)    // alice
print(user_entry.value)  // 30
```

That comment is worth internalising. The struct definition can say `key: K` because struct fields are never checked. The function signature cannot, because function parameters always are. The same letter, two different fates, decided entirely by position.

Listing 18.9: A generic wrapper with a label

```
struct Tagged<T> {
    tag: String,
    data: T
}

let measurement = Tagged { tag: "temperature", data: 23.5 }
let flag = Tagged { tag: "active", data: true }

print(measurement.tag + ": " + to_string(measurement.data))  // temperature: 23.5
print(flag.tag + ": " + to_string(flag.data))                // active: true
```

18.2.3 Generic Structs with Traits

Generic structs work with trait implementations exactly as plain ones do:

Listing 18.10: Traits on generic structs

```
struct Box<T> {
    value: T
}

trait Describable {
    fn describe(self: any) -> String
}

impl Describable for Box {
    fn describe(self: any) -> String {
        return "Box(" + to_string(self.value) + ")"
    }
}

let b = Box { value: 42 }
print(b.describe())  // Box(42)

let s = Box { value: "world" }
print(s.describe())  // Box(world)
```

The impl block says `impl Describable for Box` without type parameters—because at runtime, every `Box` instance carries the same type name regardless of what went into `value`. There is exactly one `Box::describe` in the method registry, and it serves them all.

18.3 Generic Enums and Trait Implementations

Enum payloads are the other safe harbour. The `Option` and `Result` patterns from Chapter 16 read considerably better with type parameters, and they run unchanged:

Listing 18.11: A generic Option enum

```
enum Option<T> {
    Some(T),
    None
}

let found = Option::Some(42)
let missing = Option::None

print(found)    // Option::Some(42)
print(missing)  // Option::None
```

The type parameter <T> makes the intent unmistakable: Some carries a value of type T, and None carries nothing.

18.3.1 Generic Result

Listing 18.12: A generic Result enum

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}

fn divide(a: Int, b: Int) -> Result {
    if b == 0 {
        return Result::Err("division by zero")
    }
    return Result::Ok(a / b)
}

let r = divide(10, 3)
match r {
    Result::Ok(v)  => print("result: " + to_string(v)),
    Result::Err(e) => print("error: " + e),
    _ => print("?")
}
// Output: result: 3
```

With `Result<T, E>`, the definition documents that the success payload is one type and the error payload is another. Notice that `divide` declares its return type as plain `Result`, not `Result<Int, String>`: the enum name is a real type the checker knows, so it passes; a parameterised form would only add noise the runtime must then look past.

18.3.2 Recursive Generic Enums

The `Tree` example from Section 16.7 becomes more self-documenting with a type parameter:

Listing 18.13: A generic binary tree

```
enum Tree<T> {
    Leaf(T),
    Branch(any, any)
}

fn tree_sum(t: any) -> Int {
    match t {
        Tree::Leaf(v)      => v,
        Tree::Branch(l, r) => tree_sum(l) + tree_sum(r),
        _ => 0
    }
}

let tree = Tree::Branch(
    Tree::Branch(Tree::Leaf(1), Tree::Leaf(2)),
    Tree::Leaf(3)
)
print(tree_sum(tree))  // 6
```

Branch Payloads and Self-Reference

You might notice that `Branch` uses `any` for its children rather than `Tree<T>`. Nothing resolves recursive type references inside enum payloads—nothing resolves type references inside enum payloads at all—so `any` is the honest annotation for the recursive positions.

18.4 Type Expressions

Beyond simple names like `Int` and `String`, Lattice supports a rich syntax for *type expressions*—the annotations you write after colons in parameter lists and struct fields. Unlike type parameters, most of these are backed by something real.

18.4.1 Named Types

The most basic type expression is a single name:

```
fn greet(name: String) -> String {
    return "Hello, " + name
}
```

Built-in named types include `Int`, `Float`, `Bool`, `String`, `Array`, `Map`, `Set`, `Tuple`, `Channel`, `Buffer`, `Iterator`, `Ref`, and `Fn`. Custom struct and enum names are also valid—and all of these *are* checked when the function is called.

18.4.2 Parameterised Named Types

Named types can take type arguments in angle brackets:

Listing 18.14: Parameterised type expressions

```
struct Config {
    headers: Map<String, String>,
    ports: Array<Int>,
    handler: Fn
}
```

`Map<String, String>` says "a map from strings to strings." `Array<Int>` says "an array of integers." The parser recognises the angle brackets and stores the type arguments in the AST (see `src/parser.c`, the `parse_type_expr` function). The outer name is checked at call time; the arguments inside the brackets are not looked at.

18.4.3 Array Types

Lattice supports a bracket syntax for array types:

Listing 18.15: Array type annotation

```
fn sum(nums: [Int]) -> Int {
    flux total = 0
    for n in nums {
        total = total + n
    }
    return total
}

print(sum([1, 2, 3, 4, 5]))  // 15
```

The annotation `[Int]` is shorthand for "an array whose elements are integers." It is syntactically distinct from `Array<Int>` but carries the same meaning—and, like `Array<Int>`, it is checked for array-ness and nothing more.

18.4.4 Function Types

Function types describe the signature of a callable value. They arrived in v0.3.29 and work in any annotation position:

Listing 18.16: Function type annotations

```
fn apply(f: Fn(Int) -> Int, x: Int) -> Int {
    return f(x)
}

let double = |n| { return n * 2 }
print(apply(double, 5))  // 10
```

The type `Fn(Int) -> Int` means "a function that takes an `Int` and returns an `Int`." Multi-parameter function types list all parameter types:

Listing 18.17: Multi-parameter function types

```
fn apply_binary(f: Fn(Int, Int) -> Int, a: Int, b: Int) -> Int {
    return f(a, b)
}

let add = |x, y| { return x + y }
print(apply_binary(add, 3, 4))  // 7
```

`fn` Params Require Annotations; Lambda Params Forbid Them

Look closely at `let double = |n| { return n * 2 }`. The lambda's parameter carries no type annotation—and it must not. Writing `|n: Int|` is a parse error: `parse error: expected ',', got ':'`. That is the exact inverse of `fn` declarations, where every parameter annotation is *required*.

Now that `Fn(Int) -> Int` exists as a type, these two rules sit uncomfortably close together. The way to hold them apart: you annotate the function *type* at the place that receives the closure, never the closure's own parameters at the place that builds it.

18.4.5 Combining Type Expressions

These forms compose. Here is a function that takes a transformation function and an array, and returns a new array—the classic "map":

Listing 18.18: Combining function and array types

```
fn transform<T, U>(items: [T], f: Fn(T) -> U) -> [U] {
    flux result = []
    for item in items {
        result.push(f(item))
    }
    return result
}

let names = ["alice", "bob", "charlie"]
let lengths = transform(names, |s| { return len(s) })
print(lengths)  // [5, 3, 7]
```

The signature `fn transform<T, U>(items: [T], f: Fn(T) -> U) -> [U]` is dense but readable: it takes an array of `T`, a function from `T` to `U`, and returns an array of `U`. And it runs, because every `T` and `U` in it sits inside a bracket or a `Fn(...)`—never bare.

Type Expressions as Documentation

Since Lattice does not enforce the interiors of these annotations, they serve as *machine-readable documentation*. The language server already reads them for hover information (see `src/lsp_server.c`), and the doc generator surfaces them. Write type expressions for your future self and your teammates.

18.4.6 Phase Annotations in Types

There is one part of the type syntax that is fully enforced today, and it is the part Lattice cares most about. Type expressions can carry phase qualifiers:

Listing 18.19: Phase-annotated types

```
struct Point { x: any, y: any }

fn to_crystal(p: ~Point) -> *Point {
    return freeze(p)
}

flux a = Point { x: 1, y: 2 }
print(phase_of(to_crystal(a)))  // crystal

fix locked = Point { x: 1, y: 2 }
print(phase_of(to_crystal(locked)))
// vm error: phase constraint violation in function 'to_crystal'
```

The tilde (~) denotes a fluid (mutable) value; the asterisk (*) denotes a crystal (frozen) value. Pass a crystal where the signature demands fluid and the call is refused before the body runs—the tree-walker spells it out in more detail ("parameter `'p'` requires flux argument, got crystal (fix)"), but all three backends stop you. Phase annotations are covered in depth in Chapter 11.

That contrast is the shape of Lattice's type system in miniature: phases are a real, enforced property of values, and everything else in the annotation syntax is currently a note to the reader.

18.5 Practical Patterns with Generics

With the boundary mapped, here are the patterns that use generic annotations well—all of them written so the runtime never sees a bare type parameter.

18.5.1 The Wrapper Pattern

A common need is to wrap a value with metadata:

Listing 18.20: A timestamped wrapper

```
struct Timestamped<T> {
    value: T,
    created_at: Int
}

fn stamp<T>(value: any, time: Int) -> Timestamped {
    return Timestamped { value: value, created_at: time }
}

let record = stamp("measurement: 23.5C", 1700000000)
print(record.value)       // measurement: 23.5C
print(record.created_at)  // 1700000000
```

The type parameter `T` signals that `Timestamped` is agnostic about what it wraps. It could hold a string, an integer, or another struct. The field says `value: T`; the factory parameter says `any`.

18.5.2 The Builder Pattern

Generic structs pair well with method chaining:

Listing 18.21: A generic builder

```
struct Builder<T> {
    parts: [T],
    count: Int
}

trait Buildable {
    fn add(self: any, item: any) -> any
    fn items(self: any) -> any
}

impl Buildable for Builder {
    fn add(self: any, item: any) -> any {
        flux new_parts = []
        for i in 0..self.count {
            new_parts.push(self.parts[i])
        }
        new_parts.push(item)
        return Builder { parts: new_parts, count: self.count + 1 }
    }

    fn items(self: any) -> any {
        return self.parts
    }
}

let list = Builder { parts: [], count: 0 }
let result = list.add("alpha").add("beta").add("gamma")
print(result.items())  // [alpha, beta, gamma]
```

18.5.3 Higher-Order Generic Functions

Higher-order functions are where generic annotations pay off most, and happily they are also where the annotations are safest—the type parameters live inside `[...]` and `Fn(...)`:

Listing 18.22: Generic filter and reduce

```
fn filter<T>(items: [T], predicate: Fn(T) -> Bool) -> [T] {
    flux result = []
    for item in items {
        if predicate(item) {
            result.push(item)
        }
    }
    return result
}

fn reduce<T, U>(items: [T], initial: any, combine: Fn(U, T) -> U) -> any {
    flux acc = initial
    for item in items {
        acc = combine(acc, item)
    }
    return acc
}

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

let evens = filter(nums, |n| { return n % 2 == 0 })
print(evens)  // [2, 4, 6, 8, 10]

let total = reduce(nums, 0, |acc, n| { return acc + n })
print(total)  // 55
```

Compare the two signatures. `filter` can annotate everything, because `T` never appears bare. `reduce` would like to write `initial: U` and `-> U`, and it cannot—those are the two fatal positions, so both become `any` and the `<T, U>` header carries the intent instead. Getting this wrong is the single most common way to trip over generics in Lattice:

```
fn reduce<T, U>(items: [T], initial: U, combine: Fn(U, T) -> U) -> U { ... }
print(reduce(nums, 0, |acc, n| { return acc + n }))
// vm error: function 'reduce' parameter 'initial' expects type U, got Int
//           (did you mean 'Fn'?)
```

18.5.4 Generic Option Utilities

Building on the `Option` enum from Section 16.5:

Listing 18.23: Generic Option helpers

```
enum Option<T> {
    Some(T),
    None
}

fn map_option<T, U>(opt: Option, f: Fn(T) -> U) -> Option {
    match opt {
        Option::Some(v) => Option::Some(f(v)),
        Option::None    => Option::None,
        _ => Option::None
    }
}

fn unwrap_or<T>(opt: Option, default: any) -> any {
    match opt {
        Option::Some(v) => v,
        Option::None    => default,
        _ => default
    }
}

let name = Option::Some("Alice")
let upper = map_option(name, |s| { return s.to_upper() })
print(unwrap_or(upper, "unknown"))  // ALICE

let missing = Option::None
print(unwrap_or(missing, "anonymous"))  // anonymous
```

These utility functions work with any `Option`, regardless of the payload type, because the logic depends only on the variant, not on `T`. Note that `opt: Option` is a genuine, checked annotation—`Option` is a declared enum, so passing a non-`Option` here is caught.

18.5.5 Combining Generics, Enums, and Traits

Here is a pattern that brings all three features together—a generic "result handler" that describes and processes outcomes:

Listing 18.24: Combining generics with enums and traits

```
enum Outcome<T> {
    Success(T),
    Failure(any)
}

struct Report<T> {
    outcome: Outcome,
    label: String
}

trait Summarizable {
    fn summary(self: any) -> String
}

impl Summarizable for Report {
    fn summary(self: any) -> String {
        let status = match self.outcome {
            Outcome::Success(v) => "OK: " + to_string(v),
            Outcome::Failure(e) => "FAIL: " + to_string(e),
            _ => "UNKNOWN"
        }
        return "[" + self.label + "] " + status
    }
}

let good = Report {
    outcome: Outcome::Success(200),
    label: "health check"
}

let bad = Report {
    outcome: Outcome::Failure("timeout"),
    label: "database ping"
}

print(good.summary())  // [health check] OK: 200
print(bad.summary())   // [database ping] FAIL: timeout
```

The `Report<T>` struct holds an `Outcome<T>` enum, and the `Summarizable` trait provides a polymorphic `summary` method. Every type parameter here sits on a struct field or an enum payload, so the whole arrangement is documentation the runtime never inspects.

18.6 The State of Generics in Lattice

Lattice's generic system in vo.6.4 is best described as *syntax-first*: the parser fully understands type parameters and type arguments, the AST preserves them, and nothing downstream acts on them. This is a staged approach, and we are at stage one.

What parses and runs today:

- Generic type parameter syntax on functions, structs, and enums.
- Type parameters on struct fields and enum payloads.
- Type parameters inside array types (`[T]`) and function types (`Fn(T) -> U`).
- Type arguments in type expressions (`Map<String, Int>`).
- Array type shorthand (`[Int]`).
- Phase-qualified type expressions—the one part that is enforced.

What breaks today:

- A bare type parameter as a function parameter type (`fn f<T>(x: T)`) or return type (`fn f<T>(...) -> T`). Every call raises `expects type T, got ...`.
- Trait bounds: `fn show<T: Describable>(x: any)` is a parse error (`expected ',', got ':'`).
- Type arguments at a construction site: `Box<Int> { value: 42 }` is a parse error.

What is planned for future versions:

- Teaching the runtime type checker to recognise declared type parameters—which alone would turn the fatal positions into safe ones.
- Compile-time type checking against declared parameters.
- Trait bounds on type parameters (`<T: Describable>`).
- Better error messages when types do not match at call sites. (The current "did you mean `'Fn'`?" suggestion on a type parameter is, at least, memorable.)

Generics Do Not Prevent Type Errors

Because type parameters are never substituted, they cannot catch a mismatch: passing a `String` where the author meant an `Int` produces no diagnostic at all until the value is used incorrectly at runtime. Generics in vo.6.4 buy you clarity for human readers—nothing more, and in two positions rather less.

Write your generic annotations where they are safe. They cost nothing at runtime, they make your code substantially clearer, and when the checker arrives, most of your code will already be ready for it.

18.7 Exercises

1. **Find the boundary.** Write `fn echo<T>(x: T) -> T { return x }` and call it. Record the exact error on all three backends. Now move the type parameter one position at a time—`(x: [T]) -> any`, then `(x: any) -> [T]`, then `(f: Fn(T) -> T)`—and note which forms run. Write down the rule in your own words before re-reading Section 18.1.

2. **Generic stack.** Define a `struct Stack<T>` with fields `items: [T]` and `size: Int`. Implement a `Pushable` trait with `push`, `pop`, and `peek` methods. Test it with both integers and strings. (Remember: trait method parameters are function parameters.)

3. **Option pipeline.** Write three functions: `map_option<T, U>`, `flat_map_option<T, U>` (where `f` returns an `Option`), and `or_else<T>` (provides a fallback option). Chain them to transform `Option::Some(5)` into `Option::Some("10")` (double, then convert to string).

4. **Generic key-value store.** Define `struct KVStore<K, V>` that wraps a `Map`. Implement `set`, `get`, and `has` methods. Use it to build a phone book mapping names to numbers.

5. **Type expression exploration.** Write a function whose signature uses all three type expression forms: a named type with arguments (`Map<String, Int>`), an array type (`[String]`), and a function type (`Fn(String) -> Int`). The function should accept a list of strings and a scoring function, and return a map from strings to their scores. Then break it deliberately: pass an `Int` where the `[String]` is expected, and see which of your annotations the runtime actually defends.

What's Next

With structs, enums, traits, and generics, you now have Lattice's full toolkit for structuring data and building abstractions—along with an honest map of where that toolkit is still under construction. But programs do not merely model data—they *do* things, often many things at once. In the next part we enter the world of *concurrency*: how Lattice's structured concurrency model with `scope` and `spawn` lets you run tasks in parallel while keeping your code safe and your sanity intact.

Part V

Concurrency

Chapter 19

Structured Concurrency with `scope` and `spawn`

Suppose you need to fetch three web pages at once, resize a batch of images in parallel, or crunch numbers across all your CPU cores. Most languages hand you a thread primitive and wish you luck—leaving you to track which threads are still running, which have crashed, and whether anyone remembered to join them. Lattice takes a different approach: every concurrent task is *scoped*, meaning it is born inside a block and *guaranteed* to finish before that block returns. No orphaned threads. No forgotten futures. No fire-and-forget surprises at 3 a.m.

Let's see what that looks like.

19.1 The `scope { spawn { . . . } }` Model

The two keywords you need are `scope` and `spawn`. A `scope` block creates a concurrency region; each `spawn` inside it launches a new OS thread:

Listing 19.1: Your first scope/spawn

```
scope {
  spawn {
    print("Hello from thread A")
  }
  spawn {
    print("Hello from thread B")
  }
  print("Hello from the parent")
}
print("Both spawns are finished by here")
```

Run this a few times and you will see the three messages interleaved in different orders—but the final `print` always executes *after* every spawn has completed. That is the fundamental contract: **a `scope` block does not return until every `spawn` inside it has finished**.

Structured Concurrency

Structured concurrency is a programming discipline in which concurrent tasks are confined to a lexical scope. The scope acts as a lifetime boundary: it waits for all child tasks to complete (or fail) before control flows past it. In Lattice, this is expressed with `scope { spawn { ... } }`.

19.1.1 What Goes Where

You can place any code inside a `scope` block, not only `spawn` statements. Non-spawn statements run *first*, on the parent thread, before any spawn threads are launched:

Listing 19.2: Synchronous setup inside a scope

```
scope {
  // This runs first, on the parent thread.
  let urls = ["https://example.com/a", "https://example.com/b"]
  print("Starting downloads for ${urls.len()} URLs")

  spawn {
    // Thread 1
    print("Downloading ${urls[0]}")
  }
  spawn {
    // Thread 2
    print("Downloading ${urls[1]}")
  }
}
```

The synchronous preamble is compiled into a separate body chunk and executed before the spawn threads are created. If the preamble fails, the spawns never start.

Scope Is an Expression

Like most constructs in Lattice, `scope` is an expression—but it always evaluates to `Unit`. If you need to collect results from spawned tasks, use a `Channel` or a `Ref`, both of which we will explore later in this chapter and in Chapter 20.

19.1.2 Multiple Spawns

There is no hard limit on the number of `spawn` blocks inside a single `scope`. Each one becomes its own OS thread:

Listing 19.3: Many spawns in one scope

```
scope {
  spawn { print("Worker 1") }
  spawn { print("Worker 2") }
  spawn { print("Worker 3") }
  spawn { print("Worker 4") }
}
// All four workers are done.
```

Under the hood, the compiler counts the `spawn` blocks at compile time and emits a single `OP_SCOPE` instruction that carries the spawn count as an inline operand. The VM then allocates a `VMSpawnTask` for each spawn, clones the virtual machine state, and calls `pthread_create` for each task—all visible in `src/stackvm.c`.

Keep Spawns Coarse-Grained

Each `spawn` creates a real OS thread with its own stack and cloned VM environment. Spawning thousands of micro-tasks is not the right pattern here—use iterators or channels for fine-grained parallelism. Think of `spawn` as "I have a meaningful chunk of work that should run on its own core."

19.2 Why Structured Concurrency Matters

If you have ever chased a bug involving a detached thread that outlived its parent function, you already know why structured concurrency matters. Let's make the argument concrete.

19.2.1 The Problem with Unstructured Threads

Consider pseudocode in a language with raw thread spawning:

Listing 19.4: An unstructured concurrency hazard (pseudocode)

```
// Danger: unstructured thread (NOT valid Lattice)
fn fetch_data() {
  let handle = thread_spawn(|| {
    // What if fetch_data returns before this finishes?
    // What if this panics and nobody catches it?
    let data = download("https://api.example.com/report")
    save_to_disk(data)
  })
  // Oops, we forgot to join the handle!
  return "done"
}
```

This pattern leads to three classes of bugs:

1. **Orphaned tasks.** The spawned thread continues after the function returns, possibly accessing freed memory or stale references.

2. **Silent failures.** If the thread panics, nobody is listening. Errors vanish into the void.

3. **Resource leaks.** File handles, network sockets, and memory held by the thread are never cleaned up if the parent forgets to join.

19.2.2 Lattice's Guarantee

Lattice eliminates all three by construction:

- You cannot *detach* a thread. A `spawn` outside a `scope` is not a thread at all—the compiler turns it into an ordinary inline block that runs on the current thread (see `EXPR_SPAWN` in `src/stackcompiler.c`, which emits `OP_SCOPE` with a spawn count of zero). Threads only come into existence inside a `scope`, and a `scope` always joins them.

- Every `scope` waits for *all* spawns. There is no way to "forget" to join—it is automatic.

- If *any* spawn encounters a runtime error, the scope collects it and re-raises it in the parent. The first error wins; all threads are still joined before the error surfaces.

The re-raising part deserves a closer look, because it is the one place where the three backends still disagree. In the VM implementation (see `stackvm_spawn_thread_fn` in `src/stackvm.c`), each child VM stores its error string in the `VMSpawnTask.error` field. After every thread is joined, the parent collects the first non-null error and raises it. On the default stack VM that raise leaves the interpreter loop directly, so a `try/catch` wrapped *around* the scope does not see it—the program stops with `vm error: ...` instead. On `--tree-walk` and `--regvm` the same error *is* catchable.

Catch Inside the Spawn, Not Around the Scope

As of v0.6.4, wrapping a `scope` in `try/catch` is not a portable way to handle spawn failures. Put the `try/catch` *inside* the spawn body and report the outcome through a channel. That version behaves identically on all three backends, and it gives you per-task error reporting rather than "whichever error happened to be first."

Listing 19.5: Portable error handling: catch inside the spawn

```
flux zero = 0
let outcomes = Channel::new()

scope {
  spawn {
    try {
      let result = 100 / zero
      outcomes.send("ok: ${result}")
    } catch err {
      outcomes.send("failed: ${err.message}")
    }
  }
  spawn {
    outcomes.send("I still run to completion")
  }
}

print(outcomes.recv())
print(outcomes.recv())
// Output (order varies):
// failed: division by zero
// I still run to completion
```

Notice the second spawn still finishes. When one spawn encounters an error, the others are *not* cancelled—they run to completion (or to their own errors), and only then does the scope return. If you need cancellation, use a shared `Ref` as a signal flag that spawns check periodically; we will see that pattern in Section 19.4.

19.3 Joining Semantics—No Orphaned Tasks

Let's trace exactly what happens when a `scope` executes:

1. **Environment snapshot.** The VM pushes a new scope in the environment and exports all live local variables into it via `env_define`. Each spawned thread receives a *deep clone* of this environment—spawns see the values as they existed at the moment the scope began.

2. **Synchronous body.** Any non-spawn statements run on the parent thread.

3. **Thread creation.** For each spawn, the VM calls `stackvm_clone_for_thread` to create an independent child VM, then `pthread_create` to launch the thread. Each child VM gets its own stack, heap (`DualHeap`), and garbage collector.

4. **Join barrier.** The parent calls `pthread_join` on every spawn thread in order. This is an unconditional, blocking join—the parent cannot proceed until all children are done.

5. **Cleanup.** Child VMs are freed, the environment scope is popped, and errors (if any) are propagated.

Listing 19.6: Demonstrating join semantics with timing

```
fn slow_work(label: String, ms: Int) {
  sleep(ms)
  print("${label} finished after ~${ms}ms")
}

scope {
  spawn { slow_work("Task A", 200) }
  spawn { slow_work("Task B", 100) }
  spawn { slow_work("Task C", 300) }
}
print("All tasks done")
// Output (order of first three lines varies):
// Task B finished after ~100ms
// Task A finished after ~200ms
// Task C finished after ~300ms
// All tasks done
```

The key observation: `"All tasks done"` always prints last, regardless of how fast each spawn completes.

`sleep` and `time`

`sleep(ms)` parks the calling thread for a whole number of milliseconds, and `time()` returns the current wall clock as milliseconds since the epoch. Between them they cover almost every "simulate slow work" need in this chapter—and unlike a busy-wait loop, `sleep` actually yields the core to the other spawns.

19.3.1 Nested Scopes

Scopes can nest. Each inner scope is a complete join barrier:

Listing 19.7: Nested scopes

```
scope {
  spawn {
    print("Outer spawn starts")
    scope {
      spawn { print("Inner spawn A") }
      spawn { print("Inner spawn B") }
    }
    // Inner spawns are done here.
    print("Outer spawn continues")
  }
}
print("Everything is done")
```

The inner `scope` ensures its two spawns finish before the outer spawn continues. The outer `scope` then ensures the outer spawn finishes before the final `print`. This nesting composes cleanly—you never have to reason about which thread outlives which.

19.3.2 Scopes in Loops

A common pattern is placing a `scope` inside a loop to process batches in parallel:

Listing 19.8: Scope inside a loop

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

for batch in batches {
  scope {
    spawn {
      let sum = batch.reduce(|acc, x| { acc + x }, 0)
      print("Batch sum: ${sum}")
    }
  }
}
// Batch sum: 6
// Batch sum: 15
// Batch sum: 24
```

Each batch is fully processed before the next one starts.

Each iteration's `scope` is a self-contained concurrency region. This is safe because the scope joins before the loop advances to the next iteration.

19.4 Sharing Data Between Spawned Tasks

Here is the rule you need to internalize: **each spawn gets a deep clone of the parent's variables, and nothing a spawn writes ever escapes**. Mutations inside a spawn do not affect the parent or sibling spawns—not scalars, not arrays, not maps, not anything.

Listing 19.9: Spawns get independent copies

```
flux counter = 0
flux results = [0]

scope {
  spawn {
    counter = counter + 1
    results[0] = 99
    print("Spawn A sees: ${counter}")
  }
}

print("Parent counter: ${counter}")
print("Parent results[0]: ${results[0]}")
// Spawn A sees: 1
// Parent counter: 0
// Parent results[0]: 0
```

The spawn started with `counter = 0` (cloned from the parent), incremented its own copy, and printed `1`. The parent's `counter` is still `0`, and so is `results[0]`—an index write into a captured array is as sealed off as a scalar assignment. This holds identically on all three backends.

Spawn-Capture Isolation

Every `spawn` body executes against a *deep clone* of the enclosing environment. Writes to captured variables—rebinding, index assignment, `push`, field mutation—are visible only inside that task. There is no shared-memory back channel out of a spawn. The only ways data leaves a task are a `Channel` and a `Ref`.

That last sentence is the whole design, and it is worth dwelling on. A great deal of concurrency pain in other languages comes from a variable that two threads both believe they own. Lattice removes the ambiguity by removing the sharing: if you want a value to come back out of a task, you have to say so explicitly, and the mechanism you say it with is thread-safe by construction.

Why Deep Cloning?

Deep cloning prevents data races by giving each thread its own copy of the mutable world. In the implementation (`src/stackvm.c`, `stackvm_export_locals_to_env`), the parent's live locals are cloned into the child VM's environment. Spawns can therefore mutate their local state as freely as they like without a single lock.

As of vo.6.o, frozen values are shared, not copied. If a captured variable is crystal, the spawn receives a reference-counted share of the same immutable data instead of a deep copy (Section 11.8). Nothing observable changes—a crystal cannot be mutated, so "your copy" and "our copy" behave the same—but the capture becomes an $O(1)$ pointer bump instead of an $O(n)$ traversal. Fluid values continue to be deep-cloned.

Freeze Big Captures

If a spawn only needs to *read* a large array or map, freeze it before the `scope`. The capture stops being a copy, the value stays readable inside every task, and you get a compile-time-ish guarantee that nobody meant to write to it:

```
let cities = freeze(["Seattle", "Portland", "Boston"])
let out = Channel::new()

scope {
  spawn { out.send("${cities[0]} handled") }
  spawn { out.send("${cities[1]} handled") }
}

print(out.recv())
print(out.recv())
```

19.4.1 Communicating Through Channels

If you *do* need spawns to communicate, use a `Channel`. Channels are thread-safe by design—they use a mutex-protected buffer internally (see `src/channel.c`).

Listing 19.10: Spawns communicating via a channel

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

scope {
  spawn {
    let answer = 21 * 2
    results.send(answer)
  }
  spawn {
    let answer = 6 * 7
    results.send(answer)
  }
}

// Both spawns are done; collect results.
print(results.recv())  // 42
print(results.recv())  // 42
```

Because `Channel` values are reference-counted and mutex-protected, they can be safely shared across spawn boundaries. When the parent's environment is cloned for each spawn, the channel itself is *not* deep-copied—both spawns and the parent hold references to the same underlying `LatChannel`, with the reference count incremented via `channel_retain`.

Not Every Value May Be Sent

A channel will not carry an arbitrary value out of a task. Scalars and frozen or unphased data go through; a *fluid* container does not:

```
flux tally = Map::new()
tally["hits"] = 3
results.send(tally)
// vm error: can only send crystal (frozen) values on a channel
results.send(freeze(tally))   // fine
```

The full rule—and why freezing first is also the fastest thing you can do—is Section 20.1.2.

We will explore channels in depth in Chapter 20.

19.4.2 Communicating Through Ref

For shared mutable state that doesn't need channel semantics, Lattice offers `Ref`—a reference-counted, thread-safe wrapper:

Listing 19.11: Using Ref for shared state

```
let total = Ref::new(0)

scope {
  spawn {
    let current = total.get()
    total.set(current + 10)
  }
  spawn {
    let current = total.get()
    total.set(current + 20)
  }
}

print("Total: ${total.get()}")
// Could be 10, 20, or 30 depending on timing
```

Ref Is Not Atomic

A `Ref` provides shared mutable state, but it does *not* provide atomicity. The `get()` followed by `set()` pattern above is a classic race condition. If you need atomic read-modify-write semantics, use a channel to serialize access, or restructure your code so that each spawn writes to its own channel and the parent aggregates. We will cover safe patterns in Chapter 21.

19.5 The Sublimate Phase for Thread-Safe Values

In Chapter 11, we met the **fluid** and **crystal** phases—values that are mutable or frozen, respectively. Concurrency introduces a third phase transition: **sublimated**.

The Sublimated Phase

A *sublimated* value is one that has been marked as thread-safe and immutable. Like a crystal value, a sublimated value cannot be mutated. Unlike crystal, sublimation is specifically intended for data that will cross thread boundaries. The name comes from chemistry: sublimation is the phase transition from solid directly to gas—the value "evaporates" from one thread's local world into the shared concurrent space.

19.5.1 Using `sublimate`

The `sublimate` keyword transitions a value to the sublimated phase. Like `freeze` and `thaw`, it takes its argument in parentheses, and when that argument is a plain variable name the new phase is written back into the binding:

Listing 19.12: Sublimating a value

```
flux config = [1, 2, 3]

// Mark as thread-safe before sharing
sublimate(config)
print(phase_of(config))  // sublimated

scope {
  spawn {
    // config is sublimated --- safe to read, cannot mutate
    print("Config length: ${config.len()}")
  }
  spawn {
    print("First element: ${config[0]}")
  }
}
// sublimated
// Config length: 3
// First element: 1
```

Once sublimated, any attempt to mutate the value raises a runtime error. Which error depends on how you tried:

Listing 19.13: Sublimated values cannot be mutated

```
flux data = [10, 20, 30]
sublimate(data)

data.push(40)
// vm error: cannot call mutating method 'push'
//           on a sublimated value

data[0] = 99
// vm error: cannot modify a sublimated value
```

Both messages are identical on all three backends, modulo the prefix: the default stack VM says `vm error:`, `--tree-walk` says `error:`, and `--regvm` says `regvm error:`. That prefix convention holds for every runtime error in this book, so we will quote only the message from here on.

19.5.2 Sublimate vs. Freeze

You might wonder: "If sublimated values are immutable, how are they different from frozen values?"

- `freeze` produces a **crystal** value. Crystals are immutable but are designed for single-threaded lifetime management—they participate in arenas and crystal regions.

- `sublimate` produces a **sublimated** value. Sublimated values are immutable *and* conceptually detached from any single thread's memory region. They are the right choice for data that will be read by multiple threads.

In the VM implementation (`src/stackvm.c`), both crystal and sublimated values are treated identically for mutation checks—any attempt to set a field, push to an array, or assign an index is rejected. The `OP_SUBLIMATE_VAR` opcode sets the value's phase tag to `VTAG_SUBLIMATED` and fires any reactive bonds.

Listing 19.14: Sublimate and freeze comparison

```
flux temperatures = [72.1, 68.4, 75.0]

// Option 1: freeze --- good for single-threaded immutability
fix frozen_temps = freeze(temperatures)

// Option 2: sublimate --- good for cross-thread sharing
flux shared_temps = [72.1, 68.4, 75.0]
sublimate(shared_temps)

scope {
  spawn {
    // Both are readable, neither is mutable
    print("Frozen: ${frozen_temps[0]}")
    print("Sublimated: ${shared_temps[0]}")
  }
}
// Frozen: 72.1
// Sublimated: 72.1
```

When to Sublimate

Use `sublimate` when you have configuration data, lookup tables, or shared constants that multiple spawns need to read. It communicates intent: "this value is intended for concurrent access." If you are not sharing data across threads, plain `freeze` is sufficient and more conventional.

Sublimated Values Cannot Be Sent

There is one asymmetry between the two immutable phases, and it catches people out. A crystal value may be sent on a channel; a sublimated one may not:

```
flux readings = [1, 2, 3]
sublimate(readings)
ch.send(readings)
// vm error: cannot send a sublimated value on a channel
```

Sublimation marks a value as *already* shared in place, so handing it to a channel—which copies or retains on the way through—is a contradiction the runtime refuses to resolve for you. Sublimate what spawns read directly from the captured environment; freeze what you intend to send.

19.5.3 Sublimating Complex Structures

Sublimation is deep—it applies to the entire value graph:

Listing 19.15: Deep sublimation

```
flux settings = Map::new()
settings["db_host"] = "localhost"
settings["db_port"] = 5432
settings["retries"] = 3

sublimate(settings)

scope {
  spawn {
    print("Connecting to ${settings["db_host"]}:${settings["db_port"]}")
  }
  spawn {
    print("Max retries: ${settings["retries"]}")
  }
}
// Connecting to localhost:5432
// Max retries: 3
```

After sublimation, neither the map nor any of its values can be modified—the entire structure is safe for concurrent reads.

19.6 Practical Examples

Let's put everything together with some real-world-inspired examples.

19.6.1 Parallel Computation: Fan-Out / Fan-In

The classic concurrent pattern: split work across threads, then collect results.

Listing 19.16: Fan-out/fan-in with channels

```
fn compute_chunk(numbers: Array, result_ch: Channel) {
  let sum = numbers.reduce(|acc, n| { acc + n }, 0)
  result_ch.send(sum)
}

let data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
let results = Channel::new()

// Split into 3 chunks and process in parallel
scope {
  spawn { compute_chunk(data.slice(0, 4), results) }
  spawn { compute_chunk(data.slice(4, 8), results) }
  spawn { compute_chunk(data.slice(8, 12), results) }
}

// Collect and sum the partial results
let total = results.recv() + results.recv() + results.recv()
print("Total: ${total}")  // Total: 78
```

This is a textbook fan-out/fan-in. The `scope` ensures all three workers complete before we start receiving. The channel collects partial sums, and we aggregate them on the parent thread.

19.6.2 Parallel String Processing

Listing 19.17: Normalizing names in parallel

```
fn process_name(name: String, output: Channel) {
  output.send(name.trim().to_upper())
}

let names = freeze(["  alice  ", "BOB", " Charlie ", "  dave"])
let output = Channel::new()

scope {
  spawn { process_name(names[0], output) }
  spawn { process_name(names[1], output) }
  spawn { process_name(names[2], output) }
  spawn { process_name(names[3], output) }
}

// Collect all results
flux processed_names = []
for _ in 0..4 {
  processed_names.push(output.recv())
}
print(processed_names.sort())
// [ALICE, BOB, CHARLIE, DAVE]
```

Two details worth pausing on. We froze `names` before the scope, so each spawn shares it instead of copying it. And we sorted before printing, because arrival order across four threads is anybody's guess—asserting on a set of results rather than a sequence of them is a habit worth forming early.

19.6.3 Nested Parallelism: Two-Level Fan-Out

Listing 19.18: Nested scope for two-level parallelism

```
fn process_region(region: String, results: Channel) {
  let cities = freeze(["Seattle", "Portland"])
  let city_results = Channel::new()

  scope {
    spawn {
      city_results.send("${cities[0]}: processed")
    }
    spawn {
      city_results.send("${cities[1]}: processed")
    }
  }

  let summary = "${region}: ${city_results.recv()}, ${city_results.recv()}"
  results.send(summary)
}

let results = Channel::new()

scope {
  spawn { process_region("West", results) }
  spawn { process_region("East", results) }
}

print(results.recv())
print(results.recv())
// Output (region order varies):
// West: Seattle: processed, Portland: processed
// East: Seattle: processed, Portland: processed
```

Each region processes its cities in parallel (inner `scope`), and regions themselves run in parallel (outer `scope`). The structured concurrency model guarantees that inner scopes complete before their parent spawn returns, which completes before the outer scope returns.

Freeze Aggregates Before a Nested Spawn Reads Them

The `freeze` on `cities` is not decoration. In v0.6.4, a fluid array or map that is captured *two* spawn levels deep—read inside a `spawn` that sits inside a `scope` that is itself inside a `spawn`—can tear down the running program on both bytecode VMs. Freezing the value before the inner `scope` makes the capture a shared crystal instead of a nested clone, and the pattern is stable on all three backends. Since a nested worker almost never wants to write to the outer collection anyway, freezing is the right shape as well as the safe one.

19.6.4 Using a Cancellation Flag

Since spawns cannot be forcibly cancelled, we use a shared `Ref` as a cooperative cancellation signal:

Listing 19.19: Cooperative cancellation with Ref

```
let should_stop = Ref::new(false)
let results = Channel::new()

scope {
  spawn {
    // Producer: generate values until told to stop
    flux count = 0
    while should_stop.get() == false {
      count = count + 1
      results.send(count)
      if count >= 100 {
        break
      }
    }
    results.close()
  }
  spawn {
    // Consumer: read values, signal stop after finding 42
    flux done = false
    while done == false {
      let val = results.recv()
      if val == 42 {
        print("Found 42! Signaling stop.")
        should_stop.set(true)
        done = true
      }
    }
  }
}
// Found 42! Signaling stop.
```

The `Ref` serves as a boolean flag visible to both threads. The consumer sets it when it has seen enough, and the producer checks it each iteration. This is a cooperative, structured alternative to thread cancellation.

19.6.5 Scope with Error Handling

Listing 19.20: Handling errors from parallel tasks

```
fn safe_divide(a: Int, b: Int, results: Channel) {
  if b == 0 {
    results.send("error: division by zero")
  } else {
    results.send(to_string(a / b))
  }
}

let results = Channel::new()

scope {
  spawn { safe_divide(100, 5, results) }
  spawn { safe_divide(200, 0, results) }
  spawn { safe_divide(300, 3, results) }
}

for _ in 0..3 {
  print(results.recv())
}
// Output (order may vary):
// error: division by zero
// 100
// 20
```

When errors are expected and recoverable, handling them inside the spawn and communicating results through a channel is not merely cleaner than a `try`/`catch` around the scope—as we saw in Section 19.2, on the default VM it is the only version that works.

19.7 Under the Hood

For readers who want to understand the machinery, here is a tour of how `scope` and `spawn` are compiled and executed.

19.7.1 Compilation

The compiler (`src/stackcompiler.c`) handles `scope` expressions in `EXPR_SCOPE`:

1. It counts the `spawn` blocks and separates them from the synchronous statements.

2. The synchronous statements are compiled into a *sub-body chunk*—a self-contained bytecode chunk that can be executed independently.
3. Each `spawn` body is also compiled into its own sub-body chunk.
4. A single `OP_SCOPE` instruction is emitted with the spawn count, the synchronous body index, and each spawn body index as inline operands.

If there are no `spawn` blocks (a bare `scope { ... }`), the compiler emits `OP_SCOPE` with a spawn count of zero, and the body runs synchronously as a sub-chunk—useful for isolating variable scopes.

19.7.2 Execution

When the VM encounters `OP_SCOPE` (`src/stackvm.c`):

1. It pushes a new environment scope and exports all live locals via deep cloning.
2. If `spawn_count == 0`, it runs the body chunk synchronously and returns the result.
3. Otherwise, it runs the synchronous body first, then allocates a `VMSpawnTask` array.
4. For each spawn, `stackvm_clone_for_thread` creates an independent child `StackVM` with its own runtime, heap, and GC state. The function `stackvm_export_locals_to_env` copies the parent's locals into the child's environment.
5. Each child is launched with `pthread_create`. The thread function `stackvm_spawn_thread_fn` sets up a thread-local `DualHeap`, runs the spawn chunk, and records any error.
6. The parent calls `pthread_join` on every thread, collects errors, frees child VMs, and pops the environment scope.

Thread-Local Heaps

Each spawn thread gets its own `DualHeap` for memory allocation. This means spawned tasks can allocate freely without contending on a global allocator lock. The heap is freed when the thread completes.

19.7.3 What the Runtime Guarantees

The concurrency runtime has been through several rounds of hardening, and it is worth knowing what you can lean on as of v0.6.4:

- **Spawn threads get 8 MB stacks** (`pthread_attr_setstacksize` in `src/stackvm.c`). Deeply recursive work inside a spawn behaves like deeply recursive work on the main thread.

- **Values sent on a channel are detached from the sender's heap** before they are buffered (`value_detach` in `channel_send`). A producer that finishes and tears down before its consumer drains the channel is fine—the buffered values do not live on the dead thread's heap.
- **Container-mutating builtins are serialized process-wide while any spawned thread is live.** Two sibling spawns hammering `push` on their respective clones of the same captured array will not corrupt the heap. The cost is that heavy cross-thread mutation contends on one lock, which is a fair trade for not debugging a torn array at midnight.
- **`Ref` cells carry a recursive mutex and atomic refcounts**, so `get` and `set` are individually safe even though a `get`-then-`set` pair is not atomic as a unit.
- The runtime is **clean under ThreadSanitizer with zero suppressions** (`make tsan`).

One Tree-Walker Caveat

The guarantees above are stated for the default stack VM and `--regvm`. On `--tree-walk`, sibling spawn bodies that share a captured variable can still corrupt the interpreter's environment in v0.6.4, and cross-thread `Ref` mutation has a related teardown hazard. The tree-walker remains an excellent reference implementation for reading and debugging single-threaded semantics; reach for the default backend when threads are involved.

19.8 Exercises

1. **Parallel Sum.** Given an array of 1000 integers, write a program that splits the array into 4 chunks, sums each chunk in a separate spawn, and then adds the partial sums together. Use a `Channel` to collect results.
2. **Scope Ordering.** Write a program with a scope containing three spawns that each print a message. Run it several times and observe the output order. Then add a fourth print statement *after* the scope. Verify that it always prints last.
3. **Nested Scope Hierarchy.** Create a three-level nested scope structure: an outer scope with two spawns, each of which contains an inner scope with two more spawns. Each innermost spawn should print its "address" (e.g., `"Outer 1, Inner 2"`). Verify that all messages print before the program exits.
4. **Cancellation Pattern.** Implement a producer-consumer pair where the producer generates random strings and sends them through a channel. The consumer reads strings and stops the producer (using a `Ref` flag) when it finds one longer than 10 characters.
5. **Error Propagation.** Write a scope with three spawns where the second one deliberately divides by zero. Catch the error *inside* that spawn and report it on a results channel. Confirm that the other two spawns still ran to completion by having them write to the same channel.

Then try moving the `try`/`catch` outside the `scope` and run it under all three backends—the difference is instructive.

6. **Capture Isolation.** Write a spawn that pushes ten values onto a captured `flux` array and prints the array's length from inside the task. Print the parent's length after the scope. Explain the two numbers, then rewrite the program so the parent really does receive all ten values.

What's Next

We have seen how `scope` and `spawn` give you safe, structured concurrency with automatic joining and error propagation. But we have only scratched the surface of the `Channel`—we used it as a convenient mailbox without exploring its full API. In the next chapter, we will dive deep into channels: how to create them, how `send` and `recv` work, how to multiplex across multiple channels with `select`, and how to build powerful data pipelines that transform values as they flow between threads.

Chapter 20

Channels and `select`

In the previous chapter, we used channels as convenient mailboxes—toss a value in one end, pull it out the other. But channels in Lattice are a full concurrency primitive: they are the primary mechanism for safe communication between threads, and with `select`, they become a powerful multiplexing tool that lets you wait on multiple sources of data simultaneously.

Let's start from the beginning.

20.1 `Channel::new()`, `send()`, `recv()`, `close()`

A channel is a thread-safe FIFO queue. You create one with `Channel::new()`, send values into it, and receive values from it:

Listing 20.1: Channel basics

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

ch.send(42)
ch.send("hello")
ch.send(true)

print(ch.recv())  // 42
print(ch.recv())  // hello
print(ch.recv())  // true
```

Channels are untyped in the sense that a single channel will happily carry an `Int`, then a `String`, then a `Bool`. But **not every value is eligible to be sent**—the phase system restricts which values may cross a channel, and we will get to that rule in Section 20.1.2. Values come out in the order they were sent (first-in, first-out).

Channel

A *Channel* is an unbounded, thread-safe FIFO queue. It supports three core operations: `send(value)` enqueues a value, `recv()` dequeues a value (blocking if the channel is empty), and `close()` signals that no more values will be sent. Internally, a channel is a mutex-protected vector with a condition variable for blocking receives (see `include/channel.h`).

20.1.1 `send()` — Putting Values In

The `send()` method enqueues a value into the channel's buffer:

Listing 20.2: Sending values

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

orders.send("latte")
orders.send("cappuccino")
orders.send("espresso")

print("Queued ${3} orders")
```

Sending is non-blocking: because Lattice channels are unbounded, `send()` always succeeds immediately (as long as the channel is open). Under the hood, `channel_send` in `src/channel.c` acquires the channel's mutex, pushes the value onto the internal `LatVec` buffer, signals the `cond_notempty` condition variable to wake any blocked receivers, and also wakes any select waiters.

Sending to a Closed Channel Is an Error

If you `send()` to a closed channel, the runtime raises an error:

```
let ch = Channel::new()
ch.close()
ch.send(1)
// vm error: cannot send on a closed channel
```

Before v0.5.0 the value was silently dropped and the send reported failure internally, where nobody was looking. As of v0.5.0 it is a hard error on every backend. Design your programs so that producers stop sending before the channel is closed.

20.1.2 Send Eligibility Rules

Here is where the phase system reaches into concurrency. You may have heard the old one-line rule—"only crystal values can be sent"—and as of v0.5.0 that rule is both too strict and too loose. The runtime actually asks four questions, in this order, and every one of them must come back clean:

1. **Is it sublimated?** Then no. A sublimated value is already marked as shared in place; putting it through a channel is a contradiction.

2. **Is it fluid?** Then no—*unless* it is a scalar. `Int`, `Float`, `Bool`, and `Unit` are sendable no matter what phase they carry, because there is nothing to alias.

3. **Is it unphased or crystal?** Then yes, so far. Plain data that was never phased goes through exactly like frozen data does.

4. **Does its object graph contain a closure, a `Ref`, or an iterator—at any depth, in any phase, frozen or not?** Then no. These three carry live machine state that cannot be meaningfully copied to another thread. Freezing them does not help.

A `Channel` nested inside a sent value is fine, which is what makes the request-reply pattern in Chapter 21 possible.

Listing 20.3: Send eligibility, by example

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

flux x = 7
ch.send(x)                  // OK: fluid scalar

ch.send([1, 2, 3])          // OK: unphased plain data

flux arr = [1, 2, 3]
ch.send(arr)
// vm error: can only send crystal (frozen) values on a channel
ch.send(freeze(arr))        // OK: the canonical pattern

let f = freeze([|x| { x }])
ch.send(f)
// vm error: cannot send a value containing a closure on a channel

let r = Ref::new(42)
ch.send(r)
// vm error: cannot send a value containing a Ref on a channel
```

The four messages you can get, verbatim:

- `can only send crystal (frozen) values on a channel`
- `cannot send a sublimated value on a channel`
- `cannot send a value containing a Ref on a channel`
- `cannot send a value containing a closure on a channel` (and the matching `...containing an iterator...`)

Each is prefixed `vm error:` on the default backend, `error:` under `--tree-walk`, and `regvm error:` under `--regvm`; the message text after the prefix is identical everywhere.

The First Message Undersells the Rule

The fluid-container rejection says "can only send crystal (frozen) values," which is not literally what the runtime enforces—unphased arrays and fluid integers sail right through. Read it as "this particular value needed to be frozen," not as a statement about the whole rule.

Sends are by value: the receiver gets data that is independent of anything the sender still holds. Mutate the original after sending and the receiver never notices.

Freeze Once, Send Freely

The canonical pattern—freeze a value before sending it—used to be a tax you paid for safety. Since v0.6.0 it is the fast path. A frozen aggregate lives in a refcounted shared region, so putting it on a channel is a pointer bump rather than a deep copy, no matter how large it is (Section 11.8). Streaming a frozen 10,000-element array through a channel runs roughly 9× to 14× faster than the same program forced to copy, depending on backend.

If you are about to stream a large structure repeatedly, freeze it *once* outside the loop.

20.1.3 recv() — Taking Values Out

The `recv()` method dequeues the next value from the channel:

Listing 20.4: Receiving values

```
let ch = Channel::new()
ch.send(100)
ch.send(200)

let first = ch.recv()   // 100
let second = ch.recv()  // 200
print("Got ${first} and ${second}")
```

Here is the critical behavior: **if the channel is empty and still open, `recv()` blocks the calling thread until a value is available**. This blocking behavior is what makes channels useful for synchronization—a consumer naturally waits for its producer.

Listing 20.5: Blocking recv

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

scope {
  spawn {
    // Simulate slow work
    let start = time()
    while time() - start < 100 { }
    ch.send("result ready")
  }
  spawn {
    // This recv blocks until the value arrives
    let result = ch.recv()
    print("Got: ${result}")
  }
}
```

In the implementation (`src/channel.c`, `channel_recv`), receiving works by locking the mutex and entering a `while` loop that calls `pthread_cond_wait` as long as the buffer is empty and the channel is not closed. When a value is available, it is shifted from the front of the buffer (FIFO semantics) and returned.

20.1.4 What Happens When You `recv()` from a Closed, Empty Channel?

If the channel is closed and its buffer is empty, `recv()` returns `Unit`—the "no value here" value, which prints as `()`. That is the signal a producer is done:

Listing 20.6: Receiving from a closed channel

```
let ch = Channel::new()
ch.send(1)
ch.send(2)
ch.close()

print(ch.recv())  // 1
print(ch.recv())  // 2
print(ch.recv())  // () --- channel closed and empty
```

Now for the part that trips up nearly everyone. `Unit` is not `nil`, and—crucially—`Unit` is not something you can *write*. There is no `Unit` literal and no bare `Unit` identifier in the language:

Listing 20.7: How not to test for the end of a channel

```
let done = ch.recv()

done == nil       // false! Unit and Nil are different values
done == Unit
// vm error: undefined variable 'Unit' (did you mean 'exit'?)
```

That last message is the default backend's. Under `--tree-walk` it is `error: undefined variable 'Unit'` (no suggestion), and under `--regvm` it is `regvm error: undefined variable 'Unit' (did you mean 'exit'?)`. Three wordings, one verdict: the name does not exist.

The idiom that works asks `typeof` instead—it returns the string `"Unit"`, and strings you *can* write:

Listing 20.8: The drain-loop idiom

```
flux val = ch.recv()
while typeof(val) != "Unit" {
  print("Processing: ${val}")
  val = ch.recv()
}
```

This is the shape every consumer loop in this chapter and the next one takes. Prime the variable with one `recv()` before the loop, test its type at the top, and re-`recv()` at the bottom.

`recv()` Never Returns `nil` for "Closed"

A drain loop written as `while val != nil` will never terminate—`Unit != nil` is `true`, so the loop keeps calling `recv()` on a closed channel forever, spinning at full speed. If your consumer hangs at 100% CPU right after the producer closes, this is why.

Writing `while val != Unit` instead does not rescue it: that program does not run at all, because `Unit` is an undefined variable. The two failure modes look nothing alike—one hangs, one refuses to start—but they have the same cause, which is reaching for a value the language will not let you name. Use `typeof(val) != "Unit"`.

20.1.5 `close()` — Signaling Completion

The `close()` method signals that no more values will be sent:

Listing 20.9: Closing a channel

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

tasks.send("task_a")
tasks.send("task_b")
tasks.close()

// Receivers can still drain buffered values
print(tasks.recv())  // task_a
print(tasks.recv())  // task_b
print(tasks.recv())  // ()
```

Closing is a one-way operation—once closed, a channel stays closed. The implementation (`channel_close` in `src/channel.c`) sets the `closed` flag, then broadcasts on the condition variable to wake all blocked receivers and select waiters. This ensures no thread stays blocked forever on a closed channel.

Send After Close Is an Error

As of Lattice v0.5.0, sending to a closed channel raises a runtime error on every backend (Section 20.1.2). Previously the value was silently dropped. Design your programs so that producers stop sending before closing.

Always Close Producer Channels

When a producer is done sending, close the channel. This lets consumers detect the end of the stream and exit their loops cleanly. A channel that is never closed will cause any `recv()` call to block indefinitely once the buffer is drained.

20.1.6 Channel Lifecycle

Channels are reference-counted. When you share a channel across spawns, each copy increments the reference count (`channel_retain`). When a copy goes out of scope, the count is decremented (`channel_release`). The channel's internal buffer is freed only when the last reference is released.

Listing 20.10: Channel lifecycle across spawns

```
fn producer(ch: Channel, items: Array) {
  for item in items {
    ch.send(item)
  }
  ch.close()
}

fn consumer(ch: Channel) {
  flux item = ch.recv()
  while typeof(item) != "Unit" {
    print("Consumed: ${item}")
    item = ch.recv()
  }
}

let pipeline = Channel::new()

scope {
  spawn { producer(pipeline, ["apple", "banana", "cherry"]) }
  spawn { consumer(pipeline) }
}
// Output:
// Consumed: apple
// Consumed: banana
// Consumed: cherry
```

20.2 Producer-Consumer Patterns

The producer-consumer pattern is the bread and butter of channel-based concurrency. One thread produces data, another consumes it, and the channel decouples them.

20.2.1 Single Producer, Single Consumer

The most common variant:

Listing 20.11: Single producer, single consumer

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

scope {
  // Producer
  spawn {
    for i in 0..10 {
      work_queue.send(i * i)
    }
    work_queue.close()
  }

  // Consumer
  spawn {
    flux val = work_queue.recv()
    while typeof(val) != "Unit" {
      print("Processing: ${val}")
      val = work_queue.recv()
    }
    print("Consumer done")
  }
}
```

The consumer loop is idiomatic: call `recv()`, check for `Unit` (which signals the channel is closed and drained), and process each value.

20.2.2 Multiple Producers, Single Consumer

Multiple spawns can send to the same channel:

Listing 20.12: Multiple producers, single consumer

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

scope {
  spawn {
    for i in 0..5 {
      log_channel.send("[Server A] Request ${i}")
    }
  }
  spawn {
    for i in 0..5 {
      log_channel.send("[Server B] Request ${i}")
    }
  }
  spawn {
    // Give producers a moment, then close
    let start = time()
    while time() - start < 50 { }
    log_channel.close()
  }
}

// Drain the log
flux msg = log_channel.recv()
while typeof(msg) != "Unit" {
  print(msg)
  msg = log_channel.recv()
}
```

Because the channel is mutex-protected, sends from different threads are serialized automatically—no data corruption, no lost messages.

20.2.3 Single Producer, Multiple Consumers

This pattern distributes work across consumers:

Listing 20.13: Work distribution across consumers

```
fn worker(id: Int, jobs: Channel, results: Channel) {
  flux job = jobs.recv()
  while typeof(job) != "Unit" {
    let result = "Worker ${id} processed job ${job}"
    results.send(result)
    job = jobs.recv()
  }
}

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

// Enqueue work
for i in 0..12 {
  jobs.send(i)
}
jobs.close()

scope {
  spawn { worker(1, jobs, results) }
  spawn { worker(2, jobs, results) }
  spawn { worker(3, jobs, results) }
}

// Collect all results
for _ in 0..12 {
  print(results.recv())
}
```

Each worker calls `recv()` on the same jobs channel. Because `recv()` is atomic (only one thread gets each value), the work is naturally distributed—no explicit load balancing needed.

20.3 `select` with Multiple Channels

What if you need to wait on *multiple* channels at once? Maybe you have two data sources and want to process whichever one has data ready first. This is where `select` comes in.

Listing 20.14: Basic select

```
let weather = Channel::new()
let news = Channel::new()

scope {
  spawn { weather.send("Sunny, 72F") }
  spawn { news.send("Lattice 1.0 released!") }
}

let result = select {
  report from weather => { "Weather: ${report}" }
  headline from news => { "News: ${headline}" }
}
print(result)
// Weather: Sunny, 72F
```

A `select` expression waits on multiple channels and executes the arm corresponding to the first channel that has data available. Read an arm as "bind *this name* to a value taken *from this channel*, then run this body." Both channels are loaded by the time the `select` runs here, so either arm could win; which one actually does is settled by the shuffle described in Section 20.3.2.

`select` Expression

A `select` expression monitors multiple channels simultaneously and executes the body of the first arm that successfully receives a value. If multiple channels have data ready, one is chosen at random (via Fisher-Yates shuffle in the VM) to ensure fairness. The expression evaluates to the result of the chosen arm's body.

20.3.1 Anatomy of a `select` Arm

Each arm of a `select` has three parts, in this order:

Listing 20.15: Select arm anatomy

```
select {
  binding_name from channel_expr => { body }
}
```

- `binding_name`: A variable name that will hold the received value inside the body.

- `channel_expr`: An expression that evaluates to a `Channel`, introduced by the keyword `from`.

- `body`: A brace-delimited block, introduced by `=>`, to execute when this channel delivers a value.

The Binding Name Is Not Optional

Every channel arm must name a binding, even when the body ignores the value. The parser reads the arm as *identifier*, `from`, expression, and it asks for the identifier first (`src/parser.c`, the `select` branch of `parse_primary`). Dropping it and writing `from ch => { ... }` fails before the program ever starts:

```
select {
  from ch => { "no binding" }
}
// error: parse error: expected identifier, got 'from'
```

Because this is a parse error, the wording and the bare `error:` prefix are identical on all three backends. Bind a name you do not need and leave it unused—that costs nothing.

20.3.2 Fairness

When multiple channels have data ready simultaneously, Lattice does not always pick the first arm in source order. Instead, the VM shuffles the arm indices using the Fisher-Yates algorithm before polling. This prevents starvation: a fast producer on one channel cannot permanently starve a slower producer on another.

Listing 20.16: Fairness in select

```
let fast = Channel::new()
let slow = Channel::new()

// Load both channels
for i in 0..10 {
  fast.send("fast-${i}")
  slow.send("slow-${i}")
}

// Each select picks fairly from both
for _ in 0..5 {
  let result = select {
    msg from fast => { msg }
    msg from slow => { msg }
  }
  print(result)
}
// Output:
// fast-0
// fast-1
// fast-2
// slow-0
// slow-1
```

Both arms get picked, which is the point. But run that listing twice and you will get the same five lines both times, which is not what "random" usually promises.

Fair, But Reproducible Until Something Seeds the Generator

The shuffle draws from the C library's `rand()`, and Lattice never seeds it at startup. The only `srand()` calls in the runtime are lazy ones inside `random()` and `random_int()` (`src/math_ops.c`), each guarded by a `static bool seeded`. A program that never calls either runs with the default seed, so its arm order—and therefore its output—repeats exactly, run after run, on every backend.

Add a single `random()` call before the loop and the interleaving starts moving between runs (`slow-0 fast-0 slow-1 fast-1 fast-2` was one of ours). This is worth knowing in both directions: it makes `select` behaviour reproducible while you are debugging, and it means a starvation bug can hide behind a lucky default seed.

20.3.3 Select in a Loop

A common pattern is to use `select` inside a loop to continuously process events from multiple sources:

Listing 20.17: Event loop with select

```
let keyboard = Channel::new()
let network = Channel::new()
let timer = Channel::new()

// Simulate event sources
scope {
  spawn {
    keyboard.send("key_press: Enter")
    keyboard.send("key_press: Escape")
    keyboard.close()
  }
  spawn {
    network.send("packet: 200 OK")
    network.close()
  }
  spawn {
    timer.send("tick")
    timer.send("tick")
    timer.close()
  }
}

// Process all events
flux running = true
while running {
  let event = select {
    key from keyboard => { "Keyboard: ${key}" }
    pkt from network => { "Network: ${pkt}" }
    t from timer => { "Timer: ${t}" }
    default => { running = false; "done" }
  }
  print(event)
}
// Output:
// Keyboard: key_press: Enter
// Network: packet: 200 OK
// Timer: tick
// Timer: tick
// Keyboard: key_press: Escape
// done
```

The `default` arm fires when all channels are empty (or closed), allowing us to break out of the loop cleanly. Note that `running = false` inside the arm body reaches the `flux` binding declared outside the `select`: arm bodies are compiled as sub-chunks over an exported scope, so assignments to enclosing locals are written back.

20.4 `default` and `timeout` Arms

Plain `select` blocks until one of its channel arms can receive. Two special arms modify this behavior.

20.4.1 The `default` Arm

The `default` arm executes immediately if no channel has data ready:

Listing 20.18: Select with default

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

let result = select {
  val from ch => { "Got: ${val}" }
  default => { "Nothing available" }
}
print(result)  // Nothing available
```

This turns `select` into a non-blocking check. The `default` arm is useful for polling patterns where you want to do other work when no messages are ready.

Listing 20.19: Polling pattern with default

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

scope {
  spawn {
    let start = time()
    while time() - start < 100 { }
    inbox.send("delayed message")
  }
  spawn {
    flux attempts = 0
    flux found = false
    while found == false {
      let result = select {
        msg from inbox => {
          found = true
          msg
        }
        default => {
          attempts = attempts + 1
          "waiting..."
        }
      }
      if found {
        print("Got message after ${attempts} polls: ${result}")
      }
    }
  }
}
// Output: Got message after 176511 polls: delayed message
```

That poll count is the punchline. Waiting 100 milliseconds cost us roughly 176,000 trips around the loop on the default backend (and 469,130 under `--regvm`), every one of them a full `select` setup and teardown. The exact number will differ on your machine; the order of magnitude will not.

Busy-Waiting with Default

A `select` with a `default` arm never blocks. If you put it in a tight loop, you will burn CPU cycles polling. Consider using a `timeout` arm instead if you can tolerate some latency.

20.4.2 The `timeout` Arm

The `timeout` arm takes a duration in milliseconds and fires if no channel delivers a value within that time:

Listing 20.20: Select with timeout

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

let result = select {
  data from slow_service => { "Got: ${data}" }
  timeout(500) => { "Timed out after 500ms" }
}
print(result)  // Timed out after 500ms
```

The duration goes in parentheses, unlike the channel arms. `timeout 500 => { ... }` without them is a parse error.

The timeout is implemented using `pthread_cond_timedwait` in the VM (see `src/stackvm.c`, the `OP_SELECT` handler). The VM computes an absolute deadline from the current time plus the timeout value, then waits on the shared condition variable with that deadline.

Listing 20.21: Timeout with fallback logic

```
fn fetch_with_timeout(source: Channel, timeout_ms: Int) {
  let result = select {
    data from source => { data }
    timeout(timeout_ms) => { nil }
  }
  if result == nil {
    print("Request timed out, using cached data")
    return "cached_value"
  }
  return result
}

let api = Channel::new()

// Simulate a slow API
scope {
  spawn {
    let start = time()
    while time() - start < 1000 { }
    api.send("fresh data")
  }
  spawn {
    let value = fetch_with_timeout(api, 200)
    print("Using: ${value}")
  }
}
// Output: Request timed out, using cached data
//         Using: cached_value
```

20.4.3 Combining `default` and `timeout`

You can have *either* a `default` or a `timeout` arm, but not both. If a `default` arm is present, it always takes priority over any timeout since it fires immediately when no data is ready.

Choosing Between `default` and `timeout`

- Use `default` when you want to do other work *immediately* if no message is ready (non-blocking poll).
- Use `timeout` when you can afford to wait a bit but need an upper bound on how long (bounded blocking).
- Use neither when you are happy to wait indefinitely for a message (unbounded blocking).

20.4.4 Select with All Channels Closed

If all channel arms refer to closed, empty channels:

- If a `default` arm is present, it executes.
- If a `timeout` arm is present, the select waits until the timeout, then executes the timeout arm.
- If neither is present, the select unblocks and returns `Unit`.

Listing 20.22: Select with closed channels

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

let result = select {
  val from ch => { "Got: ${val}" }
  default => { "All channels closed" }
}
print(result)  // All channels closed
```

20.5 Building Pipelines with Channels

One of the most elegant uses of channels is building data processing pipelines, where each stage reads from one channel, transforms the data, and writes to another.

20.5.1 A Two-Stage Pipeline

Listing 20.23: Two-stage channel pipeline

```
fn generate(output: Channel) {
  for i in 1..11 {
    output.send(i)
  }
  output.close()
}

fn square(input: Channel, output: Channel) {
  flux val = input.recv()
  while typeof(val) != "Unit" {
    output.send(val * val)
    val = input.recv()
  }
  output.close()
}

let stage1 = Channel::new()
let stage2 = Channel::new()

scope {
  spawn { generate(stage1) }
  spawn { square(stage1, stage2) }
}

// Consume the final output
flux result = stage2.recv()
while typeof(result) != "Unit" {
  print(result)
  result = stage2.recv()
}
// Output: 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
```

Each stage runs concurrently. As soon as the generator produces a value, the squarer can consume and transform it—the pipeline processes data in a streaming fashion rather than waiting for all data to be generated first.

20.5.2 A Three-Stage Pipeline

Let's add a filtering stage:

Listing 20.24: Three-stage pipeline: generate, filter, format

```
fn generate_numbers(output: Channel, limit: Int) {
  for i in 1..limit {
    output.send(i)
  }
  output.close()
}

fn filter_evens(input: Channel, output: Channel) {
  flux val = input.recv()
  while typeof(val) != "Unit" {
    if val % 2 == 0 {
      output.send(val)
    }
    val = input.recv()
  }
  output.close()
}

fn format_output(input: Channel, output: Channel) {
  flux val = input.recv()
  while typeof(val) != "Unit" {
    output.send("Even number: ${val}")
    val = input.recv()
  }
  output.close()
}

let raw = Channel::new()
let filtered = Channel::new()
let formatted = Channel::new()

scope {
  spawn { generate_numbers(raw, 21) }
  spawn { filter_evens(raw, filtered) }
  spawn { format_output(filtered, formatted) }
}

flux msg = formatted.recv()
while typeof(msg) != "Unit" {
  print(msg)
  msg = formatted.recv()
}
// Output:
// Even number: 2
// Even number: 4
// ...
// Even number: 20
```

20.5.3 Fan-Out Pipeline

You can split a pipeline by having multiple consumers read from the same channel:

Listing 20.25: Fan-out: two workers consuming one channel

```
fn heavy_compute(id: Int, input: Channel, output: Channel) {
  flux val = input.recv()
  while typeof(val) != "Unit" {
    // Simulate computation
    output.send("Worker ${id}: ${val * val}")
    val = input.recv()
  }
}

let input = Channel::new()
let output = Channel::new()

for i in 1..21 {
  input.send(i)
}
input.close()

scope {
  spawn { heavy_compute(1, input, output) }
  spawn { heavy_compute(2, input, output) }
}

for _ in 0..20 {
  print(output.recv())
}
```

Work is distributed across the two workers automatically because each `recv()` is atomic—only one worker gets each value.

20.6 Async Iterators: `async_iter`, `async_map`, `async_filter`

Building pipelines with raw channels is powerful but verbose. Lattice provides three built-in functions that bridge channels and iterators, giving you a more ergonomic way to build concurrent data flows.

This Section Is Default-Backend Only

Everything in this section runs on the default stack VM and nowhere else. `async_iter` is guarded at the top of `native_async_iter` (`src/runtime.c`), and the other two backends refuse it outright:

```
// --regvm
regvm error: async_iter() is only supported in the stack VM backend
// --tree-walk
error: async_iter() is only available in bytecode VM mode
```

`async_map` and `async_filter` carry no such guard. Hand either of them an ordinary iterator—`async_map(iter([1, 2, 3]), |x| x * 2)`—and it runs on all three backends. But an ordinary iterator brings no background thread with it, so what you get is a lazy `map` under a concurrent-sounding name. The concurrency in this section comes from `async_iter` alone, and that is the piece the other two backends will not run.

Everything earlier in this chapter—channels, `select`, timeouts, the pipelines in Section 20.5—runs identically on all three. If you need a concurrent pipeline that is portable across backends, build it from raw channels.

20.6.1 `async_iter` — Channel-Backed Iterators

`async_iter` takes a closure that receives a channel, spawns the closure on a background thread, and returns an iterator that yields values from that channel:

Listing 20.26: Creating an async iterator

```
let numbers = async_iter(|ch| {
  for i in 1..6 {
    ch.send(i * 10)
  }
  // Channel is automatically closed when the closure returns
})

for val in numbers {
  print(val)
}
// Output:
// 10
// 20
// 30
// 40
// 50
```

async_iter

`async_iter(producer_fn)` creates a channel, spawns a detached thread running `producer_fn` with the channel as its argument, and returns an iterator that pulls values from the channel. When the producer closure returns, the channel is automatically closed, which causes the iterator to terminate.

Under the hood (`src/runtime.c`, `native_async_iter`), the function creates a `LatChannel`, clones the current VM for thread safety, and launches a detached thread that calls the closure with the channel. The returned iterator wraps the channel and calls `channel_recv` on each `next()`.

20.6.2 async_map — Transform an Async Stream

`async_map` applies a transformation function to each element of an iterator:

Listing 20.27: Async map transformation

```
let raw_data = async_iter(|ch| {
  for i in 1..6 {
    ch.send(i)
  }
})

let doubled = async_map(raw_data, |x| x * 2)

for val in doubled {
  print(val)
}
// Output: 2, 4, 6, 8, 10
```

The transformation runs synchronously on the consumer thread as values are pulled—this is *not* an additional concurrent stage. If you need the transformation itself to run concurrently, use an explicit channel pipeline as shown in Section 20.5.

20.6.3 `async_filter` — Filter an Async Stream

`async_filter` keeps only elements that pass a predicate:

Listing 20.28: Async filter

```
let source = async_iter(|ch| {
  for i in 1..21 {
    ch.send(i)
  }
})

let evens = async_filter(source, |x| x % 2 == 0)

for val in evens {
  print(val)
}
// Output: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

20.6.4 Composing Async Operations

These functions compose naturally into a pipeline:

Listing 20.29: Composed async pipeline

```
let pipeline = async_iter(|ch| {
  for i in 1..101 {
    ch.send(i)
  }
})

let result = async_filter(
  async_map(pipeline, |x| x * x),
  |x| x % 3 == 0
)

flux count = 0
for val in result {
  count = count + 1
}
print("Found ${count} perfect squares divisible by 3")
// Output: Found 33 perfect squares divisible by 3
```

This creates a three-stage processing flow: generate numbers 1–100, square them, then keep only those divisible by 3. The generation runs on a background thread; the mapping and filtering run on the consumer thread as values are pulled through.

When to Use Async Iterators vs. Raw Channels

Use `async_iter` when you have a single producer and want to consume its output with familiar `for` loop syntax. Use raw channels when you need multiple producers, multiple consumers, bidirectional communication, or explicit `select` multiplexing.

20.7 Under the Hood: `select` Implementation

For those curious about the machinery, here is how `select` works inside the VM.

20.7.1 Compilation

The compiler (`src/stackcompiler.c`, `EXPR_SELECT`) emits a single `OP_SELECT` instruction followed by per-arm metadata:

1. **Arm count** (1 byte): how many arms the select has.
2. For each arm, four bytes:

- **Flags**: bit 0 = default, bit 1 = timeout, bit 2 = has binding name.
- **Channel/timeout index**: index of a sub-chunk constant that evaluates to the channel (or timeout value).
- **Body index**: index of the sub-chunk for the arm's body.
- **Binding index**: index of the string constant for the binding name (or `0xFF` if no binding).

20.7.2 Execution

The VM's `OP_SELECT` handler (`src/stackvm.c`) follows this algorithm:

1. Export locals to a new environment scope (so sub-chunks can see them).
2. Evaluate all channel expressions to obtain `LatChannel*` pointers.
3. Evaluate the timeout expression (if present) to get a millisecond value.
4. Build a shuffled index array (Fisher-Yates) for fairness.
5. Enter a loop:
   (a) Try a non-blocking `channel_try_recv` on each channel arm (in shuffled order).
   (b) If a value is received: bind it in the environment, execute the arm body, and break.
   (c) If all channels are closed: execute the default arm (if present) and break.
   (d) If a default arm exists: execute it immediately (non-blocking) and break.
   (e) Otherwise: register a `LatSelectWaiter` on all channels, then `pthread_cond_wait` (or `pthread_cond_timedwait` if there is a timeout).
   (f) On wake: remove waiters and retry from step (a).
6. Clean up: release channels, pop scope, push result.

The `LatSelectWaiter` structure (defined in `include/channel.h`) is a linked-list node containing a shared mutex and condition variable. When any monitored channel receives data or is closed, it signals the waiter's condition variable, waking the select loop.

20.8 Exercises

1. **Ping-Pong.** Create two channels and two spawns that play ping-pong: spawn A sends "ping" on channel 1 and waits for "pong" on channel 2; spawn B does the reverse. Run 10 rounds and print each exchange.

2. **Timeout Handler.** Write a function that takes a channel and a timeout in milliseconds. Using `select`, return the first value from the channel or the string `"timeout"` if the deadline passes. Test with both fast and slow producers.

3. **Channel Merge.** Write a function `merge` that takes two channels and returns a new channel containing all values from both, interleaved in arrival order. Use `select` in a loop with a `default` arm to detect when both channels are drained.

4. **Async Pipeline.** Using `async_iter`, `async_map`, and `async_filter`, create a pipeline that generates the first 200 natural numbers, cubes each one, filters to keep only cubes less than 10,000, and prints the results.

5. **Priority Select.** Implement a "priority queue" using two channels: a high-priority channel and a low-priority channel. Write a consumer loop that uses `select` to prefer the high-priority channel but falls through to the low-priority one when the high-priority channel is empty. (Hint: use nested `select` with `default`.)

What's Next

You now have a solid grasp of channels—creation, send/recv semantics, closing, `select` multiplexing, timeouts, and async iterators. In the next chapter, we will step back and look at the bigger picture: when to use channels versus `Ref` for shared state, how to avoid deadlocks, and a collection of battle-tested concurrency patterns including worker pools, broadcast, and request-reply.

Chapter 21

Concurrency Patterns and Pitfalls

You now know the building blocks: `scope` and `spawn` for structured concurrency, `Channel` for message passing, `select` for multiplexing, and `Ref` for shared mutable state. But knowing the primitives and knowing how to *combine* them well are different skills. This chapter is a field guide—a collection of patterns that work, pitfalls that bite, and rules of thumb earned through experience.

21.1 `Ref` for Shared Mutable State

Before we compare `Ref` with channels, let's nail down exactly what `Ref` offers. A `Ref` is a reference-counted, heap-allocated wrapper around any Lattice value. Unlike ordinary values, which are deep-cloned when passed to spawns, a `Ref` is *shared by reference*—all threads see the same underlying data.

21.1.1 `Ref::new()`, `.get()`, `.set()`

Listing 21.1: Ref basics

```
let counter = Ref::new(0)

print(counter.get())      // 0
counter.set(42)
print(counter.get())      // 42
print(counter.inner_type()) // Int
```

The API is deliberately minimal:

- `Ref::new(value)` — Creates a new `Ref` wrapping the given value. Registered as a native function (`native_ref_new` in `src/runtime.c`).
- `.get()` (or `.deref()`) — Returns a deep clone of the inner value.
- `.set(value)` — Replaces the inner value. Fails on a frozen `Ref`.
- `.inner_type()` — Returns a string describing the type of the wrapped value.

Under the hood (see `src/stackvm.c`, the `VAL_REF` case in the builtin method handler), `.get()` calls `value_deep_clone` on the inner value, giving each caller its own independent copy. The `Ref` itself is a `LatRef` struct (`include/value.h`) containing a `LatValue` and a reference count.

21.1.2 Ref Proxying

A `Ref` proxies many methods to its inner value. If the inner value is an `Array`, you can call `.push()`, `.pop()`, and `.len()` directly on the `Ref`. If it is a `Map`, you can call `.get(key)`, `.set(key, val)`, `.keys()`, and more:

Listing 21.2: Ref proxying to inner types

```
let scores = Ref::new(Map::new())

// These calls go through to the inner Map:
scores.set("Alice", 95)
scores.set("Bob", 87)
print(scores.get("Alice"))  // 95
print(scores.len())         // 2
print(scores.keys())        // [Bob, Alice]
```

Two details in that last line are worth pausing on. Printed arrays do not quote their string elements, so you get `[Bob, Alice]` rather than `["Bob", "Alice"]`. And the keys come back in the inner map's hash order, not the order we inserted them (Section 7.5.1)—the `Ref` wrapper adds no ordering of its own.

Listing 21.3: Ref with Array inner type

```
let items = Ref::new([])

items.push("hammer")
items.push("nails")
items.push("screwdriver")
print(items.len())    // 3
print(items.get())    // [hammer, nails, screwdriver]
```

This proxying makes `Ref` feel natural—you rarely need to unwrap the inner value manually.

Freezing a Ref

You can `freeze` a `Ref`, which prevents `.set()`, `.push()`, `.pop()`, and other mutating operations. The inner value itself is not frozen—only the `Ref`'s ability to be mutated through its API is locked. This is useful when you want to share read-only data without sublimating.

21.2 When to Use Channels vs. `Ref`

This is one of the most common questions in concurrent Lattice programming: "Should I use a channel or a Ref?"

The short answer: **prefer channels for communication, use Ref for shared state that does not need message-passing semantics.**

Here is a decision guide:

Scenario	Channel	Ref
Passing results from producer to consumer	✓	
Streaming data through a pipeline	✓	
Shared configuration (read-mostly)		✓
Aggregate counter (write from many threads)	✓	
Cancellation flag (boolean signal)		✓
Request-reply between two threads	✓	
Accumulating results into a collection	✓	✓*

Table 21.1: Channel vs. Ref decision matrix. *Ref works for accumulation but beware race conditions.

21.2.1 The Core Trade-Off

- **Channels** provide *ordering* and *blocking*. A `recv()` naturally waits for data, so you get synchronization for free. Channels also guarantee that each message is consumed exactly once.

- **Ref** provides *latest-value semantics*. A `.get()` always returns the current value, even if it has been overwritten multiple times. There is no ordering of writes, and no blocking—you always get an answer immediately.

Listing 21.4: Channel: ordered, exactly-once delivery

```
let ch = Channel::new()
ch.send(1)
ch.send(2)
ch.send(3)
print(ch.recv())  // 1 (always)
print(ch.recv())  // 2 (always)
print(ch.recv())  // 3 (always)
```

Listing 21.5: Ref: latest-value semantics

```
let state = Ref::new("initial")

scope {
  spawn { state.set("updated by A") }
  spawn { state.set("updated by B") }
}

print(state.get())  // updated by A -- or updated by B
```

Rule of Thumb

If you find yourself writing `ref.get()` followed by `ref.set(...)` in a spawn, you probably have a race condition. Restructure to use a channel: send partial results to an aggregator that does the read-modify-write on a single thread.

21.3 Deadlock Avoidance

A deadlock occurs when two or more threads are waiting for each other, and none can make progress. Lattice's structured concurrency model eliminates some deadlock patterns by design, but it does not eliminate all of them.

21.3.1 Classic Channel Deadlock

The most common deadlock in channel-based code: two threads each waiting to receive from the other's channel.

Listing 21.6: Deadlock: circular recv

```
// WARNING: This will deadlock!
let ch_a = Channel::new()
let ch_b = Channel::new()

scope {
  spawn {
    let val = ch_b.recv()  // Waiting for B to send
    ch_a.send(val + 1)
  }
  spawn {
    let val = ch_a.recv()  // Waiting for A to send
    ch_b.send(val + 1)
  }
}
// Neither spawn can proceed: both are blocked on recv.
```

Deadlock Is Not Detected

Lattice does not have a deadlock detector. If your program deadlocks, it will hang silently. The structured concurrency model means the `scope` will never return, and the program will appear frozen.

21.3.2 Rules for Avoiding Deadlocks

1. **Avoid circular channel dependencies.** If thread A needs a value from thread B and vice versa, redesign so that data flows in one direction, or use a mediator thread.

2. **Always close channels when done.** An unclosed channel causes `recv()` to block forever when the buffer is drained. A forgotten `close()` is the most common cause of "my program hangs" bugs.

3. **Use `select` with `timeout` for defensive waiting.** If you are not certain a value will arrive, protect yourself with a timeout:

Listing 21.7: Timeout as deadlock prevention

```
let result = select {
  val from slow_channel => { val }
  timeout(5000) => { print("Possible deadlock!"); nil }
}
```

4. **Prefer fan-in to circular patterns.** Instead of two threads exchanging messages directly, have both send to a central coordinator channel and let a single thread make decisions.

5. **Send before receive when possible.** If a thread must both send and receive, do the sends first (or in a non-blocking way) so that you are not holding a blocking receive while another thread waits on your send.

21.3.3 Deadlock-Free Pattern: The Coordinator

Listing 21.8: Coordinator pattern avoids circular waits

```
let requests = Channel::new()
let responses = Channel::new()

scope {
  // Worker A: send request, wait for response
  spawn {
    requests.send("compute_42")
    let answer = responses.recv()
    print("A got: ${answer}")
  }

  // Coordinator: receive request, send response
  spawn {
    let req = requests.recv()
    if req == "compute_42" {
      responses.send(42)
    }
    requests.close()
    responses.close()
  }
}
```

Data flows in one direction (worker → coordinator → worker), so there is no cycle.

21.4 Common Patterns

21.4.1 Worker Pool

A fixed set of workers that consume from a shared job queue. Every worker here runs the drain loop from Section 20.1.4—prime with one `recv()`, test `typeof(job) != "Unit"`, re-`recv()` at the bottom—and so does every other pattern in this chapter:

Listing 21.9: Worker pool pattern

```
fn run_worker_pool(jobs: Channel, results: Channel, num_workers: Int) {
  scope {
    // Launch workers --- use nested spawns for dynamic count
    spawn {
      scope {
        spawn {
          flux job = jobs.recv()
          while typeof(job) != "Unit" {
            results.send("W1: ${job} done")
            job = jobs.recv()
          }
        }
        spawn {
          flux job = jobs.recv()
          while typeof(job) != "Unit" {
            results.send("W2: ${job} done")
            job = jobs.recv()
          }
        }
        spawn {
          flux job = jobs.recv()
          while typeof(job) != "Unit" {
            results.send("W3: ${job} done")
            job = jobs.recv()
          }
        }
      }
      results.close()
    }
  }
}

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

// Enqueue jobs
for i in 0..9 {
  jobs.send("task_${i}")
}
jobs.close()

// Run the pool
run_worker_pool(jobs, results, 3)

// Collect results
flux r = results.recv()
while typeof(r) != "Unit" {
  print(r)
```

The key insight: because multiple workers call `recv()` on the same `jobs` channel, work is distributed automatically. When the channel is closed and drained, all workers exit their loops.

21.4.2 Broadcast

Channels are point-to-point: each value is consumed by exactly one receiver. To broadcast a value to multiple listeners, you need multiple channels:

Listing 21.10: Broadcast pattern

```
fn broadcast(value: any, listeners: Array) {
  for listener in listeners {
    listener.send(value)
  }
}

let listener_a = Channel::new()
let listener_b = Channel::new()
let listener_c = Channel::new()
let listeners = [listener_a, listener_b, listener_c]

broadcast("system shutting down", listeners)

// Each listener gets the message independently
print(listener_a.recv())  // system shutting down
print(listener_b.recv())  // system shutting down
print(listener_c.recv())  // system shutting down
```

For a real system, you might wrap this in a `Broadcaster` struct that manages the listener list and handles subscriptions:

Listing 21.11: Broadcaster with subscribers

```
fn make_broadcaster() {
  let subscribers = Ref::new([])

  let subscribe = |ch| {
    let current = subscribers.get()
    current.push(ch)
    subscribers.set(current)
  }

  let publish = |message| {
    let subs = subscribers.get()
    for sub in subs {
      sub.send(message)
    }
  }

  return [subscribe, publish]
}

let fns = make_broadcaster()
let subscribe = fns[0]
let publish = fns[1]

let ch1 = Channel::new()
let ch2 = Channel::new()
subscribe(ch1)
subscribe(ch2)

publish("hello everyone")
print(ch1.recv())  // hello everyone
print(ch2.recv())  // hello everyone
```

21.4.3 Request-Reply

A common pattern in concurrent services: a client sends a request along with a "reply channel," and the server sends the response back on that channel.

Listing 21.12: Request-reply pattern

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

scope {
  // Server: process requests
  spawn {
    flux req = server_inbox.recv()
    while typeof(req) != "Unit" {
      let query = req[0]
      let reply_ch = req[1]
      if query == "ping" {
        reply_ch.send("pong")
      } else {
        reply_ch.send("unknown: ${query}")
      }
      req = server_inbox.recv()
    }
  }

  // Client: send request with reply channel
  spawn {
    let reply = Channel::new()
    server_inbox.send(["ping", reply])
    print("Server says: ${reply.recv()}")

    let reply2 = Channel::new()
    server_inbox.send(["hello", reply2])
    print("Server says: ${reply2.recv()}")

    server_inbox.close()
  }
}
// Output:
// Server says: pong
// Server says: unknown: hello
```

Each request bundles a fresh reply channel. The server reads the request, computes the answer, and sends it back on the client's reply channel. This pattern decouples clients from the server's internal state and allows multiple clients to interact with a single server concurrently.

21.5 Testing Concurrent Code

Concurrent code is notoriously hard to test. Bugs may appear only under specific timing conditions, making them intermittent and frustrating. Here are strategies that help.

21.5.1 Deterministic Tests with Channels

The best concurrent tests eliminate timing from the equation. Use channels as synchronization points to create a deterministic order of operations:

Listing 21.13: Deterministic concurrent test

```
fn test_producer_consumer() {
  let ch = Channel::new()
  let results = Channel::new()

  scope {
    spawn {
      ch.send(10)
      ch.send(20)
      ch.send(30)
      ch.close()
    }
    spawn {
      flux sum = 0
      flux val = ch.recv()
      while typeof(val) != "Unit" {
        sum = sum + val
        val = ch.recv()
      }
      results.send(sum)
    }
  }

  let total = results.recv()
  if total != 60 {
    print("FAIL: expected 60, got ${total}")
  } else {
    print("PASS: producer-consumer sum is correct")
  }
}

test_producer_consumer()
```

The channels enforce ordering: the consumer sees all three values before computing the sum. No sleep calls, no timing assumptions.

21.5.2 Stress Testing with Repeated Runs

Some concurrency bugs only manifest under contention. Run your test many times:

Listing 21.14: Stress test via repeated runs

```
fn test_concurrent_counter_stress() {
  flux failures = 0

  for trial in 0..100 {
    let results = Channel::new()

    scope {
      spawn { results.send(1) }
      spawn { results.send(1) }
      spawn { results.send(1) }
    }

    let total = results.recv() + results.recv() + results.recv()
    if total != 3 {
      failures = failures + 1
    }
  }

  if failures > 0 {
    print("FAIL: ${failures}/100 trials failed")
  } else {
    print("PASS: all 100 trials correct")
  }
}

test_concurrent_counter_stress()
```

21.5.3 Testing with Timeouts

Protect tests from deadlocks by using `select` with a timeout:

Listing 21.15: Test with timeout protection

```
fn test_with_timeout() {
  let result_ch = Channel::new()

  scope {
    spawn {
      // Code under test
      result_ch.send("computed_value")
    }
  }

  let outcome = select {
    val from result_ch => { val }
    timeout(2000) => { "TIMEOUT" }
  }

  if outcome == "TIMEOUT" {
    print("FAIL: test timed out (possible deadlock)")
  } else if outcome == "computed_value" {
    print("PASS: got expected result")
  } else {
    print("FAIL: unexpected result: ${outcome}")
  }
}

test_with_timeout()
```

If the test deadlocks, the timeout arm fires after 2 seconds instead of hanging forever.

21.5.4 Testing Error Propagation

Here is a test you would expect to pass everywhere, and it does not. Wrap a `scope` in a `try` and raise an error inside one of its spawns:

Listing 21.16: Testing error propagation — reads differently on each backend

```
fn test_spawn_error_propagation() {
  flux caught_error = false

  try {
    scope {
      spawn {
        // Deliberately cause an error
        let arr = [1, 2, 3]
        let bad = arr[99]  // Out of bounds
      }
    }
  } catch err {
    caught_error = true
  }

  if caught_error {
    print("PASS: error from spawn was caught")
  } else {
    print("FAIL: error was not propagated")
  }
}

test_spawn_error_propagation()
```

Under `--tree-walk` and `--regvm` this prints `PASS: error from spawn was caught`. On the default stack VM it prints neither line. The spawn's error walks straight past the `catch` and ends the process:

```
vm error: array index out of bounds: 99 (len 3)
stack trace (most recent call last):
  [line 23] in <script>
  [line 5] in test_spawn_error_propagation()
```

`try` Does Not Contain a `spawn` Error on the Default Backend

A `try` wrapped around a `scope` catches errors raised by a `spawn` on `--tree-walk` and `--regvm`, and does not catch them on the default stack VM—there the error is fatal to the whole program. This is reproducible, not a race: repeated runs of the listing above give the same answer every time on each backend.

The practical consequence for tests is blunt. A test that relies on catching a spawn failure will report a pass on two backends and take your test runner down on the third, and because the process dies mid-suite you lose every result after it. Handle errors *inside* the spawn and report the outcome over a channel, which behaves identically everywhere:

```
let status = Channel::new()
scope {
  spawn {
    try {
      let arr = [1, 2, 3]
      let bad = arr[99]
      status.send("no error")
    } catch err {
      status.send("caught")
    }
  }
}
print(status.recv())  // caught
```

21.5.5 Rules of Thumb for Concurrent Tests

1. **Never use sleep for synchronization.** Use channels instead—they provide both data transfer and synchronization in one primitive.

2. **Test the contract, not the timing.** Assert on *what* values you receive, not *when* you receive them. Concurrent order is nondeterministic by design.

3. **Run stress tests in CI.** A test that passes once might fail under load. Run concurrent tests 100+ times in your continuous integration pipeline.

4. **Isolate concurrent tests.** Each test should create its own channels and spawns. Shared global state between tests is a recipe for flaky results.

5. **Use timeouts as safety nets.** Every concurrent test should have a maximum runtime to prevent CI pipelines from hanging.

The GC and Concurrency

Each spawned thread gets its own `DualHeap` and GC state, so garbage collection in one thread does not pause other threads. If you are writing stress tests that allocate heavily inside spawns, you can be confident that GC pauses are thread-local. See the test file `tests/test_gc_concurrent.lat` for examples of GC-heavy concurrent workloads.

21.6 Exercises

1. **Safe Counter.** Implement a thread-safe counter using a channel: one spawn acts as the "counter server" and processes increment/get requests sent by other spawns. Verify that the final count matches the expected total.

2. **Deadlock Detection.** Write a program that deliberately deadlocks (two spawns doing circular `recv()`). Then fix it using the coordinator pattern described in Section 21.3.

3. **Worker Pool Benchmark.** Create a worker pool with 4 workers that processes 100 jobs. Each job is a string to be reversed. Collect all results and verify that every job was processed exactly once.

4. **Broadcast System.** Implement a pub-sub system using channels: a publisher broadcasts messages to all subscribers, each subscriber counts the messages it receives, and at the end each subscriber reports its count. Verify all subscribers received the same number of messages.

What's Next

With structured concurrency, channels, select, and the patterns in this chapter, you have a complete toolkit for writing concurrent Lattice programs that are safe, composable, and testable. In the next part of the book, we shift gears from the language core to the standard library—starting with files, paths, and the filesystem in Chapter 22.

Part VI

The Standard Library

Chapter 22

Files, Paths, and the Filesystem

Every program eventually needs to talk to the outside world, and for most programs, that world is the filesystem. Whether you are reading a configuration file at startup, writing a log, or processing a batch of images, Lattice gives you a set of focused, no-surprises functions that map cleanly onto the underlying operating system. There are no deep class hierarchies to learn and no "stream" abstractions to wire together—just functions that take paths and return values.

In this chapter we will start with the three workhorses of text file I/O: `read_file`, `write_file`, and `append_file`. From there we will explore the broader `fs` family of functions for querying and manipulating the filesystem, the `path` helpers for constructing and dissecting file paths, temporary files and directories, and finally the `Buffer` type for working with raw binary data.

22.1 read_file, write_file, append_file

Let's start with the most common operation in scripting: reading a file into a string.

Listing 22.1: Reading and printing a file

```
let contents = read_file("config.toml")
print(contents)
```

`read_file` takes a path as a `String` and returns the entire file contents as a `String`. If the file does not exist or cannot be opened, it returns `nil`. Under the hood, the implementation in `src/builtins.c` opens the file with `fopen`, seeks to the end to determine the length, allocates exactly enough memory, and reads the contents in one pass.

Text In, Text Out

`read_file` is a *text* function: it treats the contents as a C string, which means it stops at the first NUL byte it encounters. Although Lattice strings themselves are binary-safe as of v0.6.4 (they can carry NUL bytes as ordinary data), `read_file` remains one of the surfaces that truncates. If you need to read arbitrary binary data—an image, a compiled bytecode file, a protocol buffer—use `read_file_bytes` instead, which returns a `Buffer` with every byte intact. We cover Buffers in Section 22.5.

Writing is the mirror operation:

Listing 22.2: Writing a string to a file

```
let report = "Scan completed: 0 issues found.\n"
write_file("report.txt", report)
```

`write_file` creates the file if it does not exist, or *truncates* it if it does. It returns `true` on success and `false` on failure. This is a full overwrite—if you need to add lines to an existing file, use `append_file`:

Listing 22.3: Appending to a log file

```
fn log_event(message: String) {
    let timestamp = time_format(time(), "%Y-%m-%d %H:%M:%S")
    let line = "[" + timestamp + "] " + message + "\n"
    append_file("app.log", line)
}

log_event("Server started on port 8080")
log_event("Accepted connection from 192.168.1.42")
```

`append_file` opens the file in append mode (`"a"` in C terms), so writes are guaranteed to go to the end of the file even if another process is writing concurrently. Like `write_file`, it creates the file if it does not exist.

write_file Truncates!

A common mistake is to use `write_file` when you meant `append_file`. If your program writes a log file using `write_file` in a loop, each iteration will obliterate everything the previous iteration wrote. When in doubt, ask yourself: "Do I want to replace or extend?"

22.1.1 A Complete Read-Modify-Write Cycle

A typical pattern combines all three operations. Suppose we have a simple counter file that stores a single integer:

Listing 22.4: Incrementing a persistent counter

```
fn increment_counter(path: String) -> Int {
    flux count = 0

    if file_exists(path) {
        let text = read_file(path)
        count = parse_int(text.trim())
    }

    count += 1
    write_file(path, to_string(count))
    return count
}

let visits = increment_counter("/tmp/visit_count.txt")
print("Visit #" + to_string(visits))
```

Notice how we check `file_exists` before reading. This is a deliberate choice—`read_file` returns `nil` on a missing file, but checking explicitly makes the intent clearer and lets us set a sensible default.

Prefer try/catch for Robust Code

For production code, wrap file operations in `try`/`catch` blocks (see Chapter 10). The examples in this chapter omit error handling for clarity, but real programs should always account for missing files, permission errors, and full disks.

22.2 The fs Module

Beyond reading and writing file contents, Lattice exposes a family of functions for querying and manipulating the filesystem itself. These are registered as top-level built-in functions—no import is required.

22.2.1 Checking Existence: file_exists, is_file, is_dir

Listing 22.5: Checking existence and type

```
let path = "/etc/hosts"

print(file_exists(path))   // true
print(is_file(path))       // true
print(is_dir(path))        // false
print(is_dir("/etc"))      // true
```

`file_exists` returns `true` if *anything* exists at the given path—file, directory, symlink, or device node. If you need to distinguish between files and directories, use `is_file` and `is_dir`. All three delegate to the POSIX `stat` system call internally (see `src/fs_ops.c`).

22.2.2 Listing Directories: list_dir

Listing 22.6: Listing directory entries

```
let entries = list_dir(".")
for entry in entries {
    print(entry)
}
```

`list_dir` returns an array of strings—one per entry in the directory, excluding the special `.` and `..` entries. The entries are file *names*, not full paths. If you need full paths, combine with `path_join`:

Listing 22.7: Building full paths from directory entries

```
let dir = "/var/log"
let entries = list_dir(dir)
for name in entries {
    let full_path = path_join(dir, name)
    let info = stat(full_path)
    print(full_path + " (" + info["type"] + ", " + to_string(info["size"]) + " bytes)")
}
```

22.2.3 Creating and Removing Directories: mkdir, rmdir

Listing 22.8: Creating and removing directories

```
mkdir("output")
write_file("output/results.csv", "name,score\nAlice,97\n")

// Later, clean up:
delete_file("output/results.csv")
rmdir("output")
```

`mkdir` creates a single directory with permissions `0755`. It does *not* create intermediate directories—if you need `mkdir -p` behavior, you will need to create each level yourself:

Listing 22.9: Creating nested directories

```
fn mkdir_p(path: String) {
    let parts = path.split("/")
    flux current = ""
    for part in parts {
        if part == "" {
            if current == "" { current = "/" }
            continue
        }
        if current == "" {
            current = part
        } else {
            current = path_join(current, part)
        }
        if !file_exists(current) {
            mkdir(current)
        }
    }
}

mkdir_p("output/reports/2024/q4")
```

One subtlety worth calling out: `path_join("", part)` produces `"/part"`—an *absolute* path—so we handle the first component of a relative path separately. A leading empty part (which `split` produces for absolute paths like `"/var/log"`) seeds `current` with the root instead.

`rmdir` removes an *empty* directory. Attempting to remove a non-empty directory will produce an error. If you need to recursively remove a directory tree, you must walk it yourself, deleting files first and directories bottom-up.

22.2.4 Globbing: glob

When you need to find files matching a pattern, `glob` is your friend:

Listing 22.10: Finding files with glob patterns

```
// Find all Lattice source files in the current directory
let sources = glob("*.lat")
for path in sources {
    print("Found: " + path)
}

// Find all JPEG images in a photos directory
let photos = glob("photos/*.jpg")
print("Found " + to_string(photos.len()) + " photos")
```

The pattern syntax follows the POSIX `glob(3)` convention: `*` matches any sequence of characters within a path component, `?` matches a single character, and `[abc]` matches character classes. On systems that support it, brace expansion (`{jpg,png,gif}`) is also available.

Listing 22.11: Brace expansion in glob patterns

```
// Find all image files (on systems with GLOB_BRACE support)
let images = glob("assets/*.{jpg,png,gif,webp}")
```

If no files match, `glob` returns an empty array—it does not treat the absence of matches as an error. You can see this behavior in `src/fs_ops.c`, where `GLOB_NOMATCH` returns `NULL` with a count of zero rather than setting the error string.

22.2.5 File Metadata: stat and file_size

The `stat` function returns a map with detailed metadata about a path:

Listing 22.12: Inspecting file metadata with stat

```
let info = stat("README.md")
print("Type: " + info["type"])           // "file"
print("Size: " + to_string(info["size"]) + " bytes")
print("Modified: " + to_string(info["mtime"]))  // epoch milliseconds
print("Mode: " + to_string(info["mode"]))        // permission bits
```

The returned map contains five keys:

Key	Type	Description
`"type"`	`String`	`"file"`, `"dir"`, `"symlink"`, or `"other"`
`"size"`	`Int`	Size in bytes
`"mtime"`	`Int`	Last modification time (epoch milliseconds)
`"mode"`	`Int`	Unix permission bits (e.g., 420, which is octal 644)
`"permissions"`	`Int`	Same as `"mode"` (alias)

If you only need the file size, `file_size` is a convenient shortcut that returns an `Int` directly:

Listing 22.13: Quick file size check

```
let bytes = file_size("database.db")
let megabytes = bytes / (1024 * 1024)
print("Database is " + to_string(megabytes) + " MB")
```

22.2.6 Copying, Renaming, and Deleting

Listing 22.14: File operations: copy, rename, delete

```
// Make a backup before modifying
copy_file("config.toml", "config.toml.bak")

// Rename (move) a file
rename("draft.md", "final.md")

// Clean up temporary files
delete_file("scratch.tmp")
```

`copy_file` reads the source in 8 KB chunks and writes them to the destination, making it safe for large files. `rename` performs an atomic rename on the same filesystem (it delegates to the C `rename()` function). `delete_file` unlinks a single file; it cannot remove directories.

22.2.7 Permissions and Canonical Paths

Listing 22.15: Changing permissions and resolving paths

```
// Make a script executable
write_file("deploy.sh", "#!/bin/sh\necho 'Deploying...'\n")
chmod("deploy.sh", 0o755)

// Resolve symlinks and relative components
let canonical = realpath("./src/../README.md")
print(canonical)  // "/home/user/project/README.md"
```

`chmod` takes a path and an integer mode. You can use octal literals like `0o755` for readability. `realpath` resolves a path to its absolute, canonical form—expanding symlinks, removing `.` and `..` components, and verifying that the path exists.

Filesystem Functions at a Glance

All filesystem functions are registered as top-level built-ins in `src/runtime.c`. They require no imports. On Emscripten (browser) builds, all filesystem operations return errors, since there is no real filesystem available—check the Emscripten stubs in `src/fs_ops.c` if you are curious about the boundary.

22.3 The path Module

File paths are strings, but they have structure. The path functions help you manipulate that structure without resorting to brittle string concatenation.

22.3.1 Joining Paths: path_join

Listing 22.16: Joining path components

```
let dir = "/home/user/projects"
let file = "lattice/main.lat"
let full = path_join(dir, file)
print(full)  // "/home/user/projects/lattice/main.lat"
```

`path_join` is variadic—you can pass any number of string arguments:

Listing 22.17: Joining multiple path segments

```
let config_path = path_join("/etc", "lattice", "config.toml")
print(config_path)  // "/etc/lattice/config.toml"
```

The implementation in `src/path_ops.c` is careful about separators. It avoids producing double slashes (`//`) when one component ends with a slash and the next begins with one, and it inserts a separator when neither component has one. This means you do not have to worry about trailing slashes on your directory paths:

Listing 22.18: path_join handles trailing slashes

```
// All of these produce the same result:
print(path_join("/tmp/", "file.txt"))   // "/tmp/file.txt"
print(path_join("/tmp", "file.txt"))    // "/tmp/file.txt"
print(path_join("/tmp/", "/file.txt"))  // "/tmp/file.txt"
```

22.3.2 Dissecting Paths: path_dir, path_base, path_ext

Three functions let you take a path apart:

Listing 22.19: Dissecting a file path

```
let path = "/home/user/photos/vacation.jpg"

print(path_dir(path))   // "/home/user/photos"
print(path_base(path))  // "vacation.jpg"
print(path_ext(path))   // ".jpg"
```

`path_dir` returns the directory portion of a path—everything before the last slash. If the path has no slash, it returns `"."` (the current directory).

`path_base` returns the filename—everything after the last slash. If the path ends with a slash, it returns an empty string.

`path_ext` returns the file extension, *including the dot*. If there is no extension, or if the filename starts with a dot (like `.gitignore`), it returns an empty string.

Listing 22.20: Edge cases in path functions

```
// No directory component
print(path_dir("readme.md"))   // "."
print(path_base("readme.md"))  // "readme.md"

// Hidden files
print(path_ext(".gitignore"))  // "" (dot-prefixed names are not extensions)

// No extension
print(path_ext("Makefile"))    // ""

// Root path
print(path_dir("/"))           // "/"
```

22.3.3 Building a File Organizer

Let's combine path functions with filesystem operations to build something practical—a script that organizes files into directories by extension:

Listing 22.21: Organizing files by extension

```
fn organize_by_extension(source_dir: String) {
    let files = list_dir(source_dir)

    for name in files {
        let full_path = path_join(source_dir, name)

        // Skip directories
        if is_dir(full_path) { continue }

        // Determine the target directory
        let ext = path_ext(name)
        // path_ext keeps the leading dot, so strip it with substring
        let category = if ext == "" {
            "no_extension"
        } else {
            ext.substring(1, ext.len())
        }

        let target_dir = path_join(source_dir, category)
        if !file_exists(target_dir) {
            mkdir(target_dir)
        }

        let dest = path_join(target_dir, name)
        rename(full_path, dest)
        print("Moved " + name + " -> " + category + "/")
    }
}

organize_by_extension("/tmp/downloads")
```

Use path Functions, Not String Slicing

It is tempting to split paths on `"/"` manually, but `path_dir`, `path_base`, and `path_ext` handle edge cases that manual string manipulation easily gets wrong: root paths, trailing slashes, dot-prefixed filenames, and empty components. Always prefer the path functions.

22.4 Temporary Files and Directories

Sometimes you need a place to stash data that should not outlive your program. Lattice provides two functions for this: `tempfile` and `tempdir`.

Listing 22.22: Creating temporary files and directories

```
// Create a temporary file
let tmp_path = tempfile()
print(tmp_path)  // e.g., "/tmp/lattice_a3kJ9x"

write_file(tmp_path, "intermediate results go here")
let data = read_file(tmp_path)
print(data)  // "intermediate results go here"

// Clean up when done
delete_file(tmp_path)
```

`tempfile` creates a new file in the system's temporary directory (typically `/tmp` on Unix or the path returned by `GetTempPathA` on Windows) with a unique name based on the template `lattice_XXXXXX`. Internally, it uses `mkstemp`, which atomically creates the file and guarantees no name collisions. The file descriptor is immediately closed, and the path is returned as a string.

`tempdir` works the same way but creates a directory instead:

Listing 22.23: Using a temporary directory as a workspace

```
let workspace = tempdir()
print(workspace)  // e.g., "/tmp/lattice_Qm7kP2"

// Use the directory for intermediate files
write_file(path_join(workspace, "step1.txt"), "Phase 1 output")
write_file(path_join(workspace, "step2.txt"), "Phase 2 output")

// List what we created
let files = list_dir(workspace)
for name in files {
    print("  " + name)
}

// Clean up
for name in files {
    delete_file(path_join(workspace, name))
}
rmdir(workspace)
```

Temporary Files Are Not Automatically Cleaned Up

Lattice does not use finalizers or RAII-style destructors to clean up temporary files. If your program exits without deleting them, they will remain on disk until the operating system's periodic cleanup runs (if ever). Always delete your temporaries explicitly, ideally using a `defer` block to ensure cleanup happens even if an error occurs:

```
let tmp = tempfile()
defer { delete_file(tmp) }
// ... work with the file ...
```

22.4.1 A Safe Processing Pipeline

A robust pattern for file processing is to write output to a temporary file, then atomically rename it to the final destination. This prevents other programs from reading a half-written file:

Listing 22.24: Atomic file writes with tempfile and rename

```
fn safe_write(destination: String, content: String) {
    let dir = path_dir(destination)
    let tmp = tempfile()
    defer {
        // Clean up the temp file if rename failed
        if file_exists(tmp) {
            delete_file(tmp)
        }
    }

    write_file(tmp, content)
    rename(tmp, destination)
}

safe_write("output/report.json", "{\"status\": \"complete\"}")
```

The `defer` block ensures the temporary file is cleaned up no matter what happens. If the `rename` succeeds, the temporary file no longer exists at its original path, so the `file_exists` check prevents a spurious error.

22.5 Working with Buffers

Strings are great for text, but the world is full of binary data: images, network packets, serialized structures, compressed archives. Lattice represents raw byte sequences with the `Buffer` type.

Buffer

A *Buffer* is a contiguous, mutable sequence of bytes (`uint8_t` values, ranging from 0 to 255). Buffers have a length (the number of bytes currently stored) and a capacity (the allocated space). You can index into a Buffer to read or write individual bytes, and use typed read/write methods to interpret sequences of bytes as integers or floating-point values.

22.5.1 Creating Buffers

There are several ways to create a Buffer:

Listing 22.25: Creating Buffers

```
// Allocate a zero-filled buffer of a given size
let zeros = Buffer::new(16)
print(zeros.len())  // 16

// Create from an array of byte values
let header = Buffer::from([0x89, 0x50, 0x4E, 0x47])
print(header.len())  // 4

// Create from a string (UTF-8 bytes)
let greeting = Buffer::from_string("Hello")
print(greeting.len())  // 5
print(greeting[0])     // 72 (ASCII 'H')
```

`Buffer::new(size)` allocates a buffer of the given size, filled with zeros. `Buffer::from(array)` creates a buffer from an array of integers, truncating each value to the range 0–255. `Buffer::from_string(str)` copies the bytes of a string into a new buffer—all of them, including any embedded NUL bytes, since Lattice strings carry an explicit length.

Buffer::new Wants a Size

As of vo.6.4 the size argument is mandatory on every backend. Calling `Buffer::new()` with no argument (or with something that is not an `Int`) raises a catchable error rather than quietly handing back an empty buffer:

```
try {
    let b = Buffer::new()
} catch e {
    print(e.message)  // "Buffer::new() expects 1 Int argument"
}
```

Earlier versions of the bytecode VM accepted the bare call and returned a zero-length buffer, which made off-by-everything bugs very hard to find.

22.5.2 Reading and Writing Binary Files

To read a file as raw bytes, use `read_file_bytes`:

Listing 22.26: Reading a binary file

```
let data = read_file_bytes("image.png")
print("File size: " + to_string(data.len()) + " bytes")

// Check the PNG magic number
if data[0] == 0x89 && data[1] == 0x50 && data[2] == 0x4E && data[3] == 0x47 {
    print("Valid PNG header!")
}
```

To write a buffer to disk, use `write_file_bytes`:

Listing 22.27: Writing binary data to a file

```
// Create a minimal BMP file header (simplified)
flux bmp = Buffer::new(0)
bmp.push(0x42)  // 'B'
bmp.push(0x4D)  // 'M'

// Write to disk
write_file_bytes("output.bmp", bmp)
```

22.5.3 Indexing and Mutation

Buffers support integer indexing for both reading and writing:

Listing 22.28: Buffer indexing

```
flux buf = Buffer::new(4)
buf[0] = 0xFF
buf[1] = 0xCA
buf[2] = 0xFE
buf[3] = 0x00

print(buf[0])  // 255
print(buf[2])  // 254

// Values are masked to a single byte
buf[0] = 256
print(buf[0])  // 0 (only the low 8 bits are kept)
```

When you assign a value to a buffer index, Lattice masks it with `0xFF`, keeping only the lowest 8 bits. Reading an index returns an `Int` in the range 0–255. Out-of-bounds access produces a runtime error.

22.5.4 The push Method and Dynamic Growth

Buffers grow dynamically when you use `push`:

Listing 22.29: Building a buffer byte by byte

```
flux packet = Buffer::new(0)
packet.push(0x01)      // version
packet.push(0x00)      // flags
packet.push_u16(1024)  // payload length as 16-bit value
packet.push_u32(42)    // sequence number as 32-bit value

print(packet.len())       // 8
print(packet.capacity())  // at least 8
```

`push` appends a single byte. `push_u16` and `push_u32` append multi-byte values in little-endian byte order.

22.5.5 Typed Read and Write Methods

When working with structured binary data, you need to read and write multi-byte values at specific offsets. Buffers provide a full suite of typed accessors:

Listing 22.30: Typed buffer reads and writes

```
flux buf = Buffer::new(20)

// Write values at specific offsets
buf.write_u8(0, 0xFF)        // 1 byte at offset 0
buf.write_u16(1, 0xCAFE)     // 2 bytes at offset 1
buf.write_u32(3, 0xDEADBEEF) // 4 bytes at offset 3

// Read them back
print(buf.read_u8(0))   // 255
print(buf.read_u16(1))  // 51966  (0xCAFE)
print(buf.read_u32(3))  // 3735928559  (0xDEADBEEF)

// 64-bit values work too
buf.write_u64(8, 123456789)
print(buf.read_u64(8))  // 123456789
buf.write_i64(8, -5)
print(buf.read_i64(8))  // -5
```

Buffers Are Not Limited to 32 Bits

The 64-bit accessors—`write_u64`, `read_u64`, `write_i64`, `read_i64`—arrived in v0.3.29. Before that, packing a 64-bit integer meant splitting it into two 32-bit halves by hand. If you find that idiom in older Lattice code, it can be collapsed into a single call now.

The full set of typed methods is:

Read	Write	Description
`read_u8(offset)`	`write_u8(offset, val)`	Unsigned 8-bit integer (1 byte)
`read_u16(offset)`	`write_u16(offset, val)`	Unsigned 16-bit integer (2 bytes)
`read_u32(offset)`	`write_u32(offset, val)`	Unsigned 32-bit integer (4 bytes)
`read_u64(offset)`	`write_u64(offset, val)`	Unsigned 64-bit integer (8 bytes)
`read_i8(offset)`	—	Signed 8-bit integer
`read_i16(offset)`	—	Signed 16-bit integer
`read_i32(offset)`	—	Signed 32-bit integer
`read_i64(offset)`	`write_i64(offset, val)`	Signed 64-bit integer
`read_f32(offset)`	—	IEEE 754 single-precision float (4 bytes)
`read_f64(offset)`	—	IEEE 754 double-precision float (8 bytes)

All multi-byte reads and writes use *little-endian* byte order, matching the native byte order on x86 and ARM systems.

Bounds Checking

The typed accessors bounds-check every offset. Reading a `u32` at offset 5 from a 7-byte buffer produces a runtime error, because the read would extend past the end of the buffer (offset 5 + 4 bytes = offset 9, which exceeds length 7):

```
flux b = Buffer::new(7)
try {
    print(b.read_u32(5))
} catch e {
    print(e.message)  // "Buffer.read_u32: index out of bounds"
}
```

The message always follows the shape `Buffer.<method>: index out of bounds`, so you can pattern-match on it if you need to. Negative offsets are rejected the same way.
One backend caveat worth knowing: on `--regvm`, only the 64-bit accessors (`read_u64`, `write_u64`, `read_i64`, `write_i64`) raise on an out-of-range offset; the narrower widths return `nil` from reads and quietly drop writes. The default VM and the tree-walking interpreter check all widths. Verify your offsets rather than relying on the error either way.

Frozen Buffers Refuse to Change

Buffers participate fully in the phase system. Once a buffer is crystallized, the mutating methods—`write_*`, `fill`, `resize`, `clear`, `push`—and index assignment all refuse:

```
flux b = Buffer::new(4)
freeze(b)
b.fill(0xFF)
// vm error: cannot call mutating method 'fill' on crystal value 'b'
//           (use thaw(b) to make it mutable)
b[0] = 1
// vm error: cannot modify a frozen value
```

Reads—`read_u8`, `len`, `slice`, `to_hex`—keep working. See Chapter 11 for the full crystallization story.

22.5.6 Slicing, Filling, and Converting

Buffers provide several utility methods for common operations:

Listing 22.31: Buffer utility methods

```
let data = Buffer::from_string("Hello, Lattice!")

// Slice: extract a sub-buffer (start, end)
let sub = data.slice(0, 5)
print(sub.to_string())  // "Hello"

// to_hex: get a hex string representation
let header = Buffer::from([0xDE, 0xAD, 0xBE, 0xEF])
print(header.to_hex())  // "deadbeef"

// to_array: convert to an array of integers
let bytes = header.to_array()
print(bytes)  // [222, 173, 190, 239]

// fill: set all bytes to a value
flux scratch = Buffer::new(8)
scratch.fill(0xFF)
print(scratch[3])  // 255

// clear: reset length to zero
scratch.clear()
print(scratch.len())  // 0

// resize: change the buffer length
flux resizable = Buffer::new(4)
resizable.resize(8)
print(resizable.len())  // 8
```

`slice(start, end)` returns a *new* buffer containing the bytes from index `start` up to (but not including) end. `to_string` interprets the buffer bytes as UTF-8 and returns a `String`. `to_hex` produces a lowercase hexadecimal representation. `to_array` returns an `Array` of `Int` values, one per byte.

22.5.7 Parsing a Binary File Format

Let's put everything together by parsing a WAV audio file header. The WAV format starts with a well-defined 44-byte header:

Listing 22.32: Parsing a WAV file header

```
fn parse_wav_header(path: String) {
    let data = read_file_bytes(path)
    if data.len() < 44 {
        print("File too small to be a valid WAV")
        return
    }

    // Bytes 0-3: "RIFF" marker
    let riff = data.slice(0, 4).to_string()
    if riff != "RIFF" {
        print("Not a RIFF file")
        return
    }

    // Bytes 4-7: file size minus 8
    let file_size = data.read_u32(4) + 8

    // Bytes 8-11: "WAVE" format
    let format = data.slice(8, 12).to_string()
    if format != "WAVE" {
        print("Not a WAVE file")
        return
    }

    // Bytes 22-23: number of channels
    let channels = data.read_u16(22)

    // Bytes 24-27: sample rate
    let sample_rate = data.read_u32(24)

    // Bytes 34-35: bits per sample
    let bits_per_sample = data.read_u16(34)

    print("WAV File: " + path)
    print("  Channels:       " + to_string(channels))
    print("  Sample rate:    " + to_string(sample_rate) + " Hz")
    print("  Bits per sample: " + to_string(bits_per_sample))
    print("  File size:      " + to_string(file_size) + " bytes")
}

parse_wav_header("recording.wav")
```

This example demonstrates the core workflow for binary file parsing: read the file as bytes, check magic numbers, and use typed reads at known offsets to extract structured fields.

Buffers Are Value Types

Like structs in Lattice, Buffers are *value types*—assigning a buffer to a new variable or passing it to a function behaves as though it made a copy. If you modify the "copy," the original is unaffected. This eliminates a whole category of aliasing bugs.

It used to also mean you paid for a byte-for-byte duplication every time a large buffer crossed a function boundary. Since v0.6.0 that is no longer true for *crystallized* buffers: freezing a buffer materializes it into a refcounted shared region once, after which every alias is a constant-time retain. The semantics do not change—a frozen buffer cannot be mutated anyway—but the cost does. If you are shuttling a multi-megabyte buffer around, `freeze` it first. For more on value semantics see Chapter 3; for the sharing model see Chapter 11.

Exercises

1. **Disk Usage Counter.** Write a function `disk_usage(dir: String) -> Int` that recursively calculates the total size in bytes of all files in a directory tree. Use `list_dir`, `is_dir`, `is_file`, and `file_size`.

2. **Find and Replace.** Write a program that takes a directory path, a search string, and a replacement string as arguments. It should find all `.txt` files in the directory using `glob`, read each one, replace all occurrences of the search string, and write the result back. Use the safe-write pattern from Section 22.4.1 to avoid corrupting files.

3. **Hex Dump.** Write a function `hex_dump(path: String, count: Int)` that reads the first `count` bytes of a file using `read_file_bytes` and prints them in a classic hex dump format: 16 bytes per line, showing both hexadecimal values and printable ASCII characters.

4. **File Deduplication.** Write a program that scans a directory, reads every file into a Buffer, computes a checksum (you can use the `to_hex` method on the bytes for a simple hash, or wait until Chapter 25 for `sha256`), and reports any duplicate files.

5. **Binary Concatenator.** Write a function that takes an array of file paths and a destination path, reads each file as bytes, and concatenates all the buffers into a single file. Pay attention to building the output buffer efficiently.

What's Next

Now that we can read and write files, the natural next question is: what *format* should those files be in? In the next chapter, we will explore Lattice's built-in support for four of the most common

data formats—JSON, TOML, YAML, and CSV—and see how to parse, generate, and round-trip structured data between them.

Chapter 23

JSON, TOML, YAML, and CSV

Programs rarely exist in isolation. They read configuration files, consume API responses, export reports, and exchange data with other programs. The format of that data matters, and there is no single format that fits every situation. JSON dominates web APIs. TOML is the configuration language of choice for many modern tools. YAML appears in CI pipelines and Kubernetes manifests. CSV is the lingua franca of spreadsheets and data science.

Lattice provides built-in support for all four. Each format has a `_parse` function that turns a string into Lattice values, and a `_stringify` function that turns Lattice values back into a string. The mapping between formats and Lattice types is consistent and predictable: JSON objects become Maps, arrays become Arrays, and scalars become the corresponding Lattice primitives.

Let's start with JSON, the format you will probably use most often.

23.1 json_parse / json_stringify

23.1.1 Parsing JSON

Listing 23.1: Parsing a JSON string

```
let text = "{\"name\": \"Lattice\", \"version\": 3, \"stable\": true}"
let data = json_parse(text)

print(data["name"])     // "Lattice"
print(data["version"])  // 3
print(data["stable"])   // true
```

`json_parse` takes a `String` and returns a Lattice value. The type mapping follows the JSON specification exactly:

JSON Type	Lattice Type
object	`Map`
array	`Array`
string	`String`
number (integer)	`Int`
number (decimal/exponent)	`Float`
`true` / `false`	`Bool`
`null`	`nil`

Notice that the parser distinguishes between integers and floats: `42` becomes an `Int`, while `42.0` or `4.2e1` becomes a `Float`. This matters because integer arithmetic in Lattice is exact, while floating-point arithmetic is subject to the usual IEEE 754 rounding.

Listing 23.2: Parsing nested JSON structures

```
let json_text = """
{
    "users": [
        {"id": 1, "name": "Alice", "active": true},
        {"id": 2, "name": "Bob", "active": false}
    ],
    "count": 2
}
"""

let data = json_parse(json_text)
let users = data["users"]

for user in users {
    let status = if user["active"] { "active" } else { "inactive" }
    print(user["name"] + " is " + status)
}
// Alice is active
// Bob is inactive
```

Error Handling in json_parse

If the input is not valid JSON, `json_parse` raises a catchable runtime error whose message carries the byte offset of the problem. The parser in `src/json.c` is a recursive-descent parser that validates strictly. Wrap parsing calls in `try`/`catch` if you are dealing with untrusted input—and if you are parsing anything that arrived over a network, read Section 23.5 before you ship.

Strings from JSON Are Binary-Safe

A JSON string may legally contain `\u0000`, and since v0.6.4 Lattice carries that through faithfully—a parsed string knows its own length rather than stopping at the first NUL byte:

```
let v = json_parse("{\"k\": \"A\\u0000B\"}")
print(v["k"].len())       // 3
print(json_stringify(v))  // {"k":"A\u0000B"}
```

Round-tripping a NUL through `json_parse` and `json_stringify` gets you the same three bytes back.

23.1.2 Stringifying to JSON

Listing 23.3: Converting Lattice values to JSON

```
let user = Map::new()
user["name"] = "Charlie"
user["age"] = 28
user["hobbies"] = ["climbing", "cooking", "Lattice"]

let json_text = json_stringify(user)
print(json_text)
// {"name":"Charlie","age":28,"hobbies":["climbing","cooking","Lattice"]}
```

`json_stringify` accepts any Lattice value that has a natural JSON representation: Maps, Arrays, Strings, Ints, Floats, Bools, and `nil`. Tuples and Buffers are serialized as JSON arrays. Refs are unwrapped automatically.

Unsupported Types

Structs, closures, channels, ranges, sets, iterators, and enums cannot be serialized to JSON directly. Attempting to stringify one of these will produce a runtime error. If you need to serialize a struct, convert it to a Map first.

The serializer handles special floating-point values gracefully: `Infinity`, `-Infinity`, and `NaN` are all encoded as `null`, since JSON has no representation for them. Strings are properly escaped, including Unicode sequences (`\uXXXX`) for control characters below `U+0020`.

Map Key Order Is Not Insertion Order

Every serializer in this chapter—`json_stringify`, `toml_stringify`, `yaml_stringify`—walks a Map's internal hash table, so the order keys come out in is determined by hashing, not by the order you inserted them. It is stable for a given set of keys, but it is not the order you wrote them in, and you should never depend on it:

```
flux m = Map::new()
m["zeta"] = 1
m["alpha"] = 2
m["mid"] = 3
print(m.keys())          // [mid, alpha, zeta]
print(json_stringify(m))  // {"mid":3,"alpha":2,"zeta":1}
```

The listings in this chapter print whatever the serializer actually emits. If you need a deterministic ordering—for a golden-file test, say, or a checksum—sort `m.keys()` and build the output yourself.

23.1.3 A Practical Example: Reading and Writing JSON Files

Listing 23.4: Loading and saving application state as JSON

```
fn load_state(path: String) -> Map {
    if !file_exists(path) {
        let defaults = Map::new()
        defaults["theme"] = "dark"
        defaults["font_size"] = 14
        defaults["recent_files"] = []
        return defaults
    }
    let text = read_file(path)
    return json_parse(text)
}

fn save_state(path: String, state: Map) {
    let text = json_stringify(state)
    write_file(path, text)
}

// Load, modify, save
flux state = load_state("settings.json")
state["font_size"] = 16
state["recent_files"] = ["main.lat", "lib/utils.lat"]
save_state("settings.json", state)
```

23.2 toml_parse / toml_stringify

TOML (Tom's Obvious Minimal Language) was designed to be a configuration file format that is easy to read and write by hand. If JSON is for machines, TOML is for humans.

23.2.1 Parsing TOML

Listing 23.5: Parsing a TOML configuration

```
let toml_text = """
title = "My Project"
version = 3

[database]
host = "localhost"
port = 5432
enabled = true

[logging]
level = "info"
file = "/var/log/app.log"
"""

let config = toml_parse(toml_text)
print(config["title"])              // "My Project"
print(config["database"]["host"])   // "localhost"
print(config["database"]["port"])   // 5432
print(config["logging"]["level"])   // "info"
```

Like `json_parse`, the TOML parser returns a Map. Table headers (`[section]`) create nested Maps. The parser supports all standard TOML features:

- Bare keys (`name = "value"`) and quoted keys (`"complex key" = "value"`)
- Dotted keys (`server.host = "localhost"`) that create nested maps automatically
- Single-quoted strings (literal, no escape processing) and double-quoted strings (with escape sequences)
- Integers (including underscored: `1_000_000`), floats, and booleans
- Inline arrays (`[1, 2, 3]`) and inline tables (`{key = "value"}`)
- Array-of-tables syntax (`[[items]]`)

Listing 23.6: TOML array of tables

```
let manifest = """
[package]
name = "my-app"
version = "1.0.0"

[[dependencies]]
name = "http"
version = "0.5.0"

[[dependencies]]
name = "json"
version = "1.2.0"
"""

let pkg = toml_parse(manifest)
print(pkg["package"]["name"])  // "my-app"

let deps = pkg["dependencies"]
for dep in deps {
    print(dep["name"] + " v" + dep["version"])
}
// http v0.5.0
// json v1.2.0
```

Each `[[dependencies]]` block pushes a new Map onto an array under the `"dependencies"` key. This is how TOML represents arrays of structured data.

23.2.2 Stringifying to TOML

Listing 23.7: Generating TOML from Lattice values

```
let config = Map::new()
config["title"] = "Generated Config"
config["debug"] = false

let server = Map::new()
server["host"] = "0.0.0.0"
server["port"] = 8080
config["server"] = server

let output = toml_stringify(config)
print(output)
// debug = false
// title = "Generated Config"
//
// [server]
// port = 8080
// host = "0.0.0.0"
```

The TOML serializer in `src/toml_ops.c` uses a two-pass approach: first it emits all scalar key-value pairs for the current table, then it recurses into sub-tables and arrays of tables. That two-pass structure is what guarantees valid TOML: every bare `key = value` pair has to appear before the first `[section]` header, or it would silently be swallowed into that section. Within each pass the keys come out in hash order, as described above.

toml_stringify Requires a Map

Unlike `json_stringify`, which accepts any serializable value, `toml_stringify` requires its argument to be a Map. TOML documents are always rooted in a table (a Map), so passing an Array or a String raises `toml_stringify: value must be a Map`.

23.3 yaml_parse / yaml_stringify

YAML is the most flexible of the four formats. It uses indentation for structure (like Python), supports both block and flow styles, and has extensive scalar type detection.

23.3.1 Parsing YAML

Listing 23.8: Parsing a YAML document

```
let yaml_text = """
name: Lattice
version: 3
features:
  - phases
  - structured concurrency
  - pattern matching
database:
  host: localhost
  port: 5432
  ssl: true
"""

let data = yaml_parse(yaml_text)
print(data["name"])               // "Lattice"
print(data["database"]["host"])   // "localhost"
print(data["database"]["ssl"])    // true

for feature in data["features"] {
    print("- " + feature)
}
// - phases
// - structured concurrency
// - pattern matching
```

Lattice's YAML parser handles the most commonly used subset of YAML:

- Block mappings (`key: value`)
- Block sequences (`- item`)
- Flow sequences (`[a, b, c]`) and flow mappings (`{a: 1, b: 2}`)
- Quoted and unquoted strings
- Document separators (`---`)

Automatic Scalar Detection

The YAML parser automatically detects scalar types from unquoted values. The words `true`, `True`, `yes`, and `Yes` all become `Bool true`. Similarly, `false`, `no`, and their capitalized variants become `false`. The words `null`, `Null`, and `~` become `nil`. Numeric-looking strings are parsed as `Int` or `Float`. If none of these patterns match, the value remains a `String`. You can see this logic in the `yaml_detect_scalar` function in `src/yaml_ops.c`.

Listing 23.9: YAML scalar auto-detection

```
let yaml_text = """
string_val: hello
int_val: 42
float_val: 3.14
bool_val: true
null_val: null
yes_val: yes
"""

let data = yaml_parse(yaml_text)
print(typeof(data["string_val"]))  // "String"
print(typeof(data["int_val"]))     // "Int"
print(typeof(data["float_val"]))   // "Float"
print(typeof(data["bool_val"]))    // "Bool"
print(typeof(data["null_val"]))    // "Nil"
print(typeof(data["yes_val"]))     // "Bool"
```

Force Strings with Quotes

If you need a value like `yes` or `3.14` to remain a string rather than being auto-detected as a boolean or float, wrap it in quotes in your YAML source: `value: "yes"` or `pi: "3.14"`.

23.3.2 Stringifying to YAML

Listing 23.10: Generating YAML from Lattice values

```
let deployment = Map::new()
deployment["apiVersion"] = "apps/v1"
deployment["kind"] = "Deployment"

let metadata = Map::new()
metadata["name"] = "my-app"
metadata["labels"] = Map::new()
metadata["labels"]["app"] = "my-app"
deployment["metadata"] = metadata

let yaml_out = yaml_stringify(deployment)
print(yaml_out)
//
// metadata:
//   name: my-app
//   labels:
//     app: my-app
// kind: Deployment
// apiVersion: apps/v1
```

The YAML serializer produces indented block-style output, nesting one level per two spaces. Two details of the emitted text are worth noticing before you diff it against something: a non-empty document opens with a newline, and a key that introduces a nested block is written with a trailing space (`"metadata: "`). Both are harmless to any YAML reader, but they will show up if you compare bytes.

Strings that could be misinterpreted as YAML special values are quoted automatically—a string containing `"true"` is emitted as `"true"` rather than bare `true`. Empty maps and arrays use the compact flow forms `{}` and `[]`.

`yaml_stringify` accepts Maps and Arrays. Anything else raises `yaml_stringify: value must be a Map or Array`.

23.4 csv_parse / csv_stringify

CSV is the oldest and most universal data exchange format. It has no formal specification (though RFC 4180 comes close), but the basic idea is universal: rows separated by newlines, fields separated by commas.

23.4.1 Parsing CSV

Listing 23.11: Parsing CSV data

```
let csv_text = "name,age,city\nAlice,30,Portland\nBob,25,Seattle\n"
let rows = csv_parse(csv_text)

for row in rows {
    print(row)
}
// [name, age, city]
// [Alice, 30, Portland]
// [Bob, 25, Seattle]
```

(Printing an Array shows its elements without quotes—every one of those fields really is a `String`.)

`csv_parse` returns an Array of Arrays—each inner array represents one row, and each element is a `String`. Unlike JSON and YAML, CSV has no type system, so all values come back as strings. If you need numbers, use `parse_int` or `parse_float` on the individual fields.

The parser handles the important edge cases:

- **Quoted fields**: Fields enclosed in double quotes can contain commas, newlines, and other special characters.
- **Escaped quotes**: A double quote inside a quoted field is escaped by doubling it: `"He said ""hello"""` produces `He said "hello"`.
- **Windows line endings**: Both `\n` and `\r\n` are handled.

Listing 23.12: Parsing CSV with quoted fields

```
let csv_text = """
name,bio,score
Alice,"Enjoys hiking, reading",95
Bob,"Said ""hello"" to everyone",87
"""

let rows = csv_parse(csv_text)
print(rows[1][1])  // Enjoys hiking, reading
print(rows[2][1])  // Said "hello" to everyone
```

23.4.2 Working with Headers

Since CSV does not distinguish between header rows and data rows, you will often want to convert the raw arrays into a more convenient structure:

Listing 23.13: Converting CSV rows to maps

```
fn csv_to_maps(rows: Array) -> Array {
    let headers = rows[0]
    flux results = []

    flux i = 1
    while i < rows.len() {
        let row = rows[i]
        let record = Map::new()
        flux j = 0
        while j < headers.len() {
            if j < row.len() {
                record[headers[j]] = row[j]
            }
            j += 1
        }
        results.push(record)
        i += 1
    }
    return results
}

let csv_text = "name,age,city\nAlice,30,Portland\nBob,25,Seattle\n"
let records = csv_to_maps(csv_parse(csv_text))

for record in records {
    print(record["name"] + " lives in " + record["city"])
}
// Alice lives in Portland
// Bob lives in Seattle
```

23.4.3 Stringifying to CSV

Listing 23.14: Generating CSV output

```
let data = [
    ["product", "price", "quantity"],
    ["Widget", "9.99", "100"],
    ["Gadget", "24.99", "50"],
    ["Doohickey", "4.99", "200"]
]

let csv_text = csv_stringify(data)
print(csv_text)
// product,price,quantity
// Widget,9.99,100
// Gadget,24.99,50
// Doohickey,4.99,200
```

`csv_stringify` takes an Array of Arrays and produces a CSV string. Each inner array becomes a row. Fields that contain commas, double quotes, or newlines are automatically wrapped in double quotes, with internal quotes escaped by doubling.

Listing 23.15: CSV auto-quoting of special characters

```
let data = [
    ["name", "description"],
    ["Widget", "A small, useful device"],
    ["Gadget", "Says \"beep\" loudly"]
]

let csv_text = csv_stringify(data)
print(csv_text)
// name,description
// Widget,"A small, useful device"
// Gadget,"Says ""beep"" loudly"
```

23.5 Strictness, Limits, and Hostile Input

Every parser in this chapter is a place where data you did not write meets memory you did allocate. A config file is usually friendly. A request body is not. Lattice's parsers therefore reject a good deal more than the letter of each specification requires, and they all fail loudly rather than guessing.

23.5.1 JSON Says No

Three rejections in particular will surprise anyone arriving from a permissive JSON library. All of them raise catchable errors, and all of them behave identically on the default VM, `--tree-walk`, and `--regvm`:

Listing 23.16: JSON inputs that Lattice refuses

```
fn attempt(label: String, text: String) {
    try {
        let v = json_parse(text)
        print(label + " -> parsed")
    } catch e {
        print(label + " -> " + e.message)
    }
}

attempt("duplicate", "{\"a\":1,\"a\":2}")
// duplicate -> json_parse error at position 10: duplicate object key

attempt("huge", "99999999999999999999999999")
// huge -> json_parse error at position 26: integer out of signed 64-bit range

attempt("trailing", "{\"a\":1} junk")
// trailing -> json_parse error at position 8: unexpected trailing content
```

Duplicate keys are the interesting one. Most JSON libraries take the last value and move on; JavaScript, Python, and Go all do. As of v0.6.4 Lattice treats a repeated key as a malformed document, because "last one wins" is exactly the ambiguity that lets an attacker show one value to your validator and a different value to your business logic. If you have code that relied on last-key-wins, it now needs a real merge step.

Integers get the same treatment. A number too large for a signed 64-bit integer is an error rather than a silently wrapped or saturated value. Floating-point numbers are still parsed as `Float`, so `1e400` is not affected by this rule.

Errors Carry Positions

Every `json_parse` failure names a byte offset: `json_parse error at position N: <reason>`. The TOML parser uses the same shape. When you are debugging a 40 KB payload, that number is worth a great deal more than "invalid JSON".

The parser also requires well-formed, shortest-form UTF-8 throughout, rejects NUL bytes inside object *keys*, and decodes `\uXXXX` surrogate pairs into correct UTF-8 rather than mangling them.

23.5.2 Depth Limits

A document like `[[[[[...]]]]]` costs the attacker one byte per level and costs a recursive-descent parser one stack frame per level. That trade is the whole of a classic denial-of-service attack, and left unchecked it ends in a stack overflow rather than an error you can catch.

Since v0.4.0 the JSON, TOML, and YAML parsers all cap nesting at 1000 levels:

Listing 23.17: Nesting past the limit is an error, not a crash

```
let bomb = "[".repeat(1100) + "]".repeat(1100)

try {
    let v = json_parse(bomb)
} catch e {
    print(e.message)
    // json_parse error at position 1000: maximum nesting depth exceeded
}
```

TOML reports the same condition as `toml_parse error at position N: maximum nesting depth exceeded`; YAML reports a bare `maximum nesting depth exceeded`. A thousand levels is far past anything a real document needs—the deepest structure in a Kubernetes manifest is single digits—so in practice the limit only ever fires on input that was trying to hurt you.

Catch at the Boundary

The pattern that scales is to wrap the parse, not the whole handler:

```
let payload = try {
    json_parse(request_body)
} catch e {
    print("rejected: " + e.message)
    nil
}
if payload == nil { return http.json(400, error_body) }
```

That way a malformed body becomes a 400 instead of a stack trace.

23.5.3 Other Guarded Surfaces

The same defensive posture shows up in the neighbouring conversion functions, which you will often reach for in the same breath as a parser:

Function	Guard
`base64_decode`	Rejects malformed padding and stray characters: `base64_decode("a===")` raises `base64_decode: invalid character in input`
`regex_match` etc.	Pattern capped at 8192 bytes and subject at 16 MB; exceeding either raises `regex: pattern or input too large`
`to_int`	A `Float` outside the `Int` range (including `NaN`) raises `to_int(): Float value out of Int range`

All three of those became errors in the v0.6.x line. Code that previously received a quietly wrong answer now receives an exception, which is a better kind of bad day.

23.6 Round-Tripping Data Between Formats

One of the advantages of Lattice's consistent parsing model is that data flows naturally between formats. Parse from one format, stringify to another. The intermediate representation—Maps, Arrays, and primitives—is the same regardless of which format you started with.

Listing 23.18: Converting TOML configuration to JSON

```
let toml_text = read_file("config.toml")
let config = toml_parse(toml_text)

// Convert to JSON for an API or another tool
let json_text = json_stringify(config)
write_file("config.json", json_text)
print("Converted TOML to JSON")
```

Listing 23.19: Converting YAML to TOML

```
let yaml_text = """
server:
  host: 0.0.0.0
  port: 8080
  workers: 4
database:
  url: postgres://localhost/mydb
  pool_size: 10
"""

let config = yaml_parse(yaml_text)
let toml_text = toml_stringify(config)
write_file("config.toml", toml_text)
print(toml_text)
// [server]
// port = 8080
// workers = 4
// host = "0.0.0.0"
//
// [database]
// pool_size = 10
// url = "postgres://localhost/mydb"
```

The sections and their contents survive the trip; only the ordering within each table reflects the Map's hashing rather than the YAML source.

23.6.1 CSV to JSON

CSV-to-JSON conversion is particularly common when preparing data for web APIs:

Listing 23.20: Converting CSV to JSON

```
let csv_text = read_file("employees.csv")
let rows = csv_parse(csv_text)

// First row is headers
let headers = rows[0]
flux records = []

flux i = 1
while i < rows.len() {
    let row = rows[i]
    let record = Map::new()
    flux j = 0
    while j < headers.len() {
        let field = if j < row.len() { row[j] } else { "" }
        // Try to parse numeric fields
        if headers[j] == "age" || headers[j] == "salary" {
            record[headers[j]] = parse_int(field)
        } else {
            record[headers[j]] = field
        }
        j += 1
    }
    records.push(record)
    i += 1
}

let json_text = json_stringify(records)
write_file("employees.json", json_text)
```

Lossy Round-Trips

Not all round-trips are lossless. TOML has no `null` type, so a JSON `null` will be lost or turned into an empty string when converted to TOML. YAML's auto-detection means that the string `"true"` in JSON might become a boolean if you round-trip through YAML without quoting. CSV loses all type information entirely—everything becomes strings. Be aware of these edge cases when converting between formats.

23.7 Building Configuration-Driven Programs

Let's apply what we have learned to build a realistic, configuration-driven application. We will build a log file analyzer that reads its configuration from a TOML file, processes log data from a CSV, and outputs results in JSON.

23.7.1 The Configuration File

Listing 23.21: analyzer.toml — configuration file

```
// Suppose analyzer.toml contains:
// [input]
// path = "logs/access.csv"
// delimiter = ","
//
// [filters]
// min_status = 400
// methods = ["GET", "POST"]
//
// [output]
// format = "json"
// path = "reports/errors.json"
// pretty = true
```

23.7.2 The Analyzer

Listing 23.22: A configuration-driven log analyzer

```
fn load_config(path: String) -> Map {
    let text = read_file(path)
    return toml_parse(text)
}

fn analyze_logs(config: Map) {
    // Read input
    let input_path = config["input"]["path"]
    let csv_text = read_file(input_path)
    let rows = csv_parse(csv_text)

    if rows.len() < 2 {
        print("No data to analyze")
        return
    }

    let headers = rows[0]
    let min_status = config["filters"]["min_status"]
    let allowed_methods = config["filters"]["methods"]

    // Find column indices
    flux status_col = -1
    flux method_col = -1
    flux path_col = -1
    flux i = 0
    while i < headers.len() {
        if headers[i] == "status" { status_col = i }
        if headers[i] == "method" { method_col = i }
        if headers[i] == "path" { path_col = i }
        i += 1
    }

    // Filter and collect matching rows
    flux errors = []
    i = 1
    while i < rows.len() {
        let row = rows[i]
        let status = parse_int(row[status_col])
        let method = row[method_col]

        if status >= min_status {
            flux method_match = false
            for allowed in allowed_methods {
                if method == allowed {
                    method_match = true
                    break
```

This program demonstrates the power of combining formats: TOML for human-readable configuration, CSV for tabular data ingestion, and JSON for structured output.

23.7.3 Using the dotenv Library

For programs that need environment-variable configuration (common in twelve-factor apps and containerized deployments), Lattice ships with a `dotenv` library that parses `.env` files:

Listing 23.23: Loading environment variables from a .env file

```
import "lib/dotenv" as dotenv

// Load .env from the current directory
dotenv.load()

// Access environment variables
let db_url = env("DATABASE_URL")
let secret = env("SECRET_KEY")
print("Connecting to: " + db_url)
```

The dotenv library, found in `lib/dotenv.lat`, supports double-quoted values with escape sequences, single-quoted literal values, inline comments, the `export` prefix, multiline values, and even variable expansion (`${VAR}` syntax). It is a good example of a real-world configuration parser built entirely in Lattice.

Layered Configuration

A robust configuration strategy combines multiple sources with clear precedence: environment variables override `.env` files, which override `config.toml` defaults. The dotenv library's `load_opts` function respects this pattern—by default it does *not* override variables that are already set in the environment.

Exercises

1. **JSON Pretty Printer.** Write a function `json_pretty(value: any) -> String` that produces indented, human-readable JSON output. Use recursion to handle nested maps and arrays, adding 2 spaces of indentation per level.

2. **Config Merger.** Write a function that takes two TOML configuration strings, parses both, and deep-merges them—the second configuration's values override the first. Nested maps should be merged recursively rather than replaced wholesale.

3. **CSV Column Filter.** Write a program that reads a CSV file, keeps only the columns specified in a TOML configuration file, and writes the filtered CSV to a new file. The config should look like: `columns = ["name", "email", "department"]`.

4. **Schema Validator.** Write a function that takes a parsed JSON value and a "schema" (a Map describing expected types for each key) and returns an array of validation errors. For example, the schema `\{"name": "String", "age": "Int"\}` should report an error if `name` is missing or `age` is a string.

5. **Format Converter CLI.** Write a program that reads from standard input in one format and writes to standard output in another. Accept the input and output formats as command-line arguments: `clat convert.lat --from yaml --to json < input.yaml`.

What's Next

We can now read files and parse structured data. The next frontier is communicating over the network. In the next chapter, we will explore Lattice's HTTP client for making web requests, TCP and TLS networking for lower-level communication, and how to build a working HTTP server—combining everything we have learned about strings, data formats, and concurrency.

Chapter 24

Networking and HTTP

Most modern programs need to talk to the network. Whether you are fetching data from a REST API, scraping a web page, or building a microservice, networking is the backbone. Lattice provides networking at two levels: a high-level HTTP client that handles the protocol details for you, and a low-level TCP/TLS layer that gives you direct control over sockets. On top of these primitives, the `http_server` library offers an Express-style framework for building web servers entirely in Lattice.

We will start at the top—making HTTP requests—and then work our way down to raw sockets, TLS, and server construction.

24.1 The http Module

The HTTP functions are available as top-level built-ins (`http_get`, `http_post`, `http_request`). They handle URL parsing, connection management, TLS negotiation for HTTPS, request formatting, chunked transfer decoding, and response parsing—all behind a single function call.

The same three functions are also bundled into a built-in module you can import under a name of your choosing:

Listing 24.1: The built-in http module

```
import "http" as h

let r = h.request("GET", "https://httpbin.org/get", Map::new())
print(r["status"])  // 200

// h.get collides with the built-in Map.get method, so bind it first
let fetch = h.get
let page = fetch("https://httpbin.org/get")
print(page["status"])  // 200
```

`h.get(url)` Returns Nil

A module is a Map, and `get` is a built-in Map method. Writing `h.get("http://...")` therefore calls *Map's* `get`, which looks up the key `"http://..."`, finds nothing, and hands you `nil`—no error, no request. Binding the member to a variable first (`let fetch = h.get`) sidesteps the collision. `h.post` and `h.request` have no such clash. The top-level `http_get` is never affected.

24.1.1 Simple GET Requests

Listing 24.2: Making an HTTP GET request

```
let response = http_get("https://httpbin.org/get")

print(response["status"])   // 200
print(response["body"])     // JSON response body

// Response headers are a Map with lowercase keys
let headers = response["headers"]
print(headers["content-type"])  // "application/json"
```

`http_get` takes a URL string and returns a `Map` with three keys:

Key	Type	Description
`"status"`	`Int`	HTTP status code (200, 404, 500, etc.)
`"body"`	`String`	Response body as a string
`"headers"`	`Map`	Response headers (keys are lowercase)

Those three keys are the whole response. There is no status-text key, no redirect history, and no separate length field—`response.keys()` prints exactly `[body, headers, status]`.

The URL must start with `http://` or `https://`. If the scheme is `https`, the HTTP layer automatically performs a TLS handshake with certificate verification before sending the request—you do not need to manage TLS yourself. Anything else is rejected up front:

Listing 24.3: An unsupported scheme never reaches the network

```
try { http_get("ftp://example.com/") }
catch e { print(e.message) }
// invalid URL: must start with http:// or https://
```

Response Bodies Are Text, Not Bytes

Lattice strings are binary-safe, but the HTTP client is not: the response body is handed to you as a C string, so it stops at the first NUL byte. Fetch a five-byte body of `41 42 00 43 44` and `response["body"].len()` reports 2, on all three backends. The raw socket layer behaves the same way (Section 24.2), so dropping down to TCP does not rescue you. For images, archives, or any other binary payload, ask the server for a text-safe encoding such as Base64, or shell out to a dedicated downloader with `exec_argv` (Section 25.4.5) and read the file back into a `Buffer`.

Listing 24.4: Fetching and parsing JSON from an API

```
let response = http_get("https://api.example.com/users")

if response["status"] == 200 {
    let users = json_parse(response["body"])
    for user in users {
        print(user["name"] + " (" + user["email"] + ")")
    }
} else {
    print("Request failed with status " + to_string(response["status"]))
}
```

HTTP and the Filesystem Chapter

Notice how naturally HTTP combines with the data format functions from Chapter 23. The response body is always a `String`; use `json_parse`, `yaml_parse`, or the other parsers to turn it into structured data.

24.1.2 POST Requests

Listing 24.5: Sending a POST request with a JSON body

```
let payload = Map::new()
payload["username"] = "alice"
payload["email"] = "alice@example.com"

let options = Map::new()
options["body"] = json_stringify(payload)

let response = http_post("https://httpbin.org/post", options)
print(response["status"])  // 200

let echoed = json_parse(response["body"])
print(echoed["data"])
// {"email":"alice@example.com","username":"alice"}
```

`http_post` takes a URL and an optional options Map. The options Map is read for three keys:

- `"body"` — the request body as a String
- `"headers"` — a Map of additional request headers
- `"timeout"` — request timeout in milliseconds (default: 30 000)

Only `"body"` Travels on Every Backend

As of v0.6.4 the bytecode backends read *only* the `"body"` key out of the options Map. The `"headers"` and `"timeout"` keys are honoured under `--tree-walk` and silently dropped by the default stack VM and by `--regvm`. Round-trip a custom header through a server that echoes it back and the difference is plain:

```
let opts = Map::new()
opts["headers"] = Map::new()
opts["headers"]["X-Handbook"] = "present"

let r = http_request("GET", "https://httpbin.org/headers", opts)
print(r["body"].contains("X-Handbook"))
// false  (default VM), false (--regvm), true (--tree-walk)
```

Until this is harmonized, do not build a client that depends on custom request headers—put what you need in the body, the URL, or the path, or open the socket yourself with the `tcp_*` functions and write the request line by hand.

24.1.3 The General-Purpose http_request

For methods beyond GET and POST—PUT, PATCH, DELETE, HEAD—use the general-purpose `http_request`:

Listing 24.6: Using http_request for other methods

```
// PUT request to update a resource
let update = Map::new()
update["name"] = "Alice Updated"

let options = Map::new()
options["body"] = json_stringify(update)

let response = http_request("PUT", "https://httpbin.org/put", options)
print(response["status"])  // 200

// DELETE request -- an empty options Map is still a Map::new()
let del = http_request("DELETE", "https://httpbin.org/delete", Map::new())
print(del["status"])  // 200
```

`http_request` takes the HTTP method as a string, the URL, and an optional options Map with the same keys as `http_post`.

Lattice Has No Map Literals

There is no `{"key": value}` literal syntax—writing one is a parse error (`parse error: unexpected token ':' in expression`). Every options Map, every JSON payload, and every response object you build by hand starts life as `Map::new()` followed by index assignment. This is still true in v0.6.4; map, set, and buffer literals remain on the list of unimplemented syntax.

Timeouts

The HTTP layer uses a default timeout of 30 seconds (`src/http.c`). A `"timeout"` key in the options Map overrides it—but the value is in milliseconds and is converted to whole seconds, with a floor of one second, so `500` and `1200` both come out as a one-second timeout. A timeout of 0 means "use the default," not "wait forever."

`http_get` takes *only* a URL—passing a second argument raises `http_get() expects (url: String)` on every backend. If you need a custom timeout, reach for `http_request("GET", url, opts)`, and remember from the warning above that the bytecode backends currently ignore the key.

24.1.4 Error Handling in HTTP

HTTP functions raise a runtime error when something goes wrong at the *network* level: DNS resolution failure, connection refused, TLS certificate validation failure, or timeout. HTTP-level errors (4xx, 5xx status codes) are *not* runtime errors—they are indicated by the `"status"` field in the response. This means you should always check the status code:

Listing 24.7: Robust HTTP error handling

```
fn fetch_user(user_id: Int) -> Any {
    let url = "https://httpbin.org/status/" + to_string(user_id)

    let response = try {
        http_get(url)
    } catch error {
        print("Network error: " + error.message)
        return nil
    }

    if response["status"] != 200 {
        print("API error: " + to_string(response["status"]))
        return nil
    }

    return json_parse(response["body"])
}
```

Two details in that listing are load-bearing.

First, the return type is `Any`, not `Map`. Return annotations are checked at run time in v0.6.4, and a function declared `-> Map` that hands back `nil` dies on the spot:

Listing 24.8: A return-type mismatch is a runtime error

```
fn fetch_user(user_id: Int) -> Map { return nil }
fetch_user(1)
// vm error: function 'fetch_user' return type expects Map, got Nil
```

The stack VM and `--tree-walk` name the function; `--regvm` prints the shorter `return type expects Map, got Nil`. If a function is allowed to fail, say so in its signature.

Second, the caught value is an error *object*, not a string. Reach for `error.message` when you want the text; `to_string(error)` gives you the whole record—message, line, and stack—which is useful for a log file and noisy for a user-facing message.

The exact wording of a network-level failure comes from the socket layer, so it is worth knowing what the common ones look like:

Cause	`error.message`
Bad scheme	`invalid URL: must start with http:// or https://`
DNS failure	`tcp_connect: nodename nor servname provided, or not known`
Nothing listening	`tcp_connect: connect: Connection refused`
Bad certificate	`tls_connect: SSL_connect: error:0A000086: SSL routines::certificate verify failed`

All four are identical on the default VM, `--tree-walk`, and `--regvm`; only the prefix the interpreter puts in front of an *uncaught* error differs (`vm error:`, `error:`, and `regvm error:` respectively).

HTTPS Certificate Verification

Lattice's TLS layer verifies server certificates against the system's trusted certificate store. Point `http_get` at a host with an expired, self-signed, or otherwise invalid certificate and the request fails with the `certificate verify failed` message above rather than returning a response. This is the correct, safe default—do not work around it in production.

24.1.5 What the Client Refuses to Send

Between v0.4.0 and v0.6.4 the HTTP client grew a set of guards that turn what used to be a security bug into a plain, catchable error. The most visible one: carriage returns, line feeds, and other control characters are rejected *before* the request line is assembled, so a stray newline in a path can no longer smuggle a second request onto the wire.

Listing 24.9: A control character in the path is refused

```
let bad = "https://httpbin.org/get\r\nX-Injected: 1"

try { http_get(bad) }
catch e { print(e.message) }
// invalid request: control characters in method, host, or path
```

The check runs after the connection is established but before a single byte of the request is written, so you see it on any reachable host. It covers the method too—`http_request("GE\r\nT", url, Map::new())` raises the identical message on all three backends. Under `--tree-walk`, where custom headers are actually transmitted, a control character in a header value raises `invalid header: control characters not allowed`.

Two more limits are worth carrying in your head. A single response is capped at 256 MiB—see `HTTP_MAX_RESPONSE_SIZE` in `src/http.c`. A server that promises more, or that keeps streaming past the cap, fails the call instead of exhausting memory. And chunked transfer decoding grows its buffer defensively, so a malformed chunk header is an error rather than a crash.

Errors, Not Sanitization

Notice what the client does *not* do: it never strips the offending characters and carries on. Rejecting the request is the honest outcome— silently rewriting a URL would hide the bug that produced it. Validate and percent-encode untrusted path segments yourself with `url_encode` (Section 24.5) before they reach `http_get`.

24.2 The net Module: TCP Networking

Beneath the HTTP layer, Lattice exposes raw TCP socket operations. These functions work with integer *socket descriptors*—opaque handles that identify a connection. The net layer tracks which descriptors are valid sockets, so passing a random integer will produce a clear error rather than undefined behavior.

24.2.1 Connecting to a Server

Listing 24.10: Connecting to a TCP server

```
let fd = tcp_connect("example.com", 80)

// Send an HTTP request manually
let request = "GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n"
tcp_write(fd, request)

// Read the response
let response = tcp_read(fd)
print(response)

tcp_close(fd)
```

`tcp_connect` performs DNS resolution and establishes a TCP connection. It returns an integer socket descriptor. `tcp_write` sends data over the connection, handling partial writes internally. `tcp_read` reads up to 8 KB of available data and returns it as a string. An empty string indicates end-of-file (the remote side closed the connection).

The Socket Layer Is NUL-Terminated

Lattice strings carry an explicit length and happily hold embedded NUL bytes—but `tcp_read`, `tcp_read_bytes`, and `tcp_write` all fall back on C string semantics, so a NUL ends the data as far as the socket is concerned. Build the five-byte string `41 42 00 43 44` from a `Buffer` and hand it to `tcp_write`:

```
let buf = Buffer::new(5)
buf.write_u8(0, 65)  // A
buf.write_u8(1, 66)  // B
buf.write_u8(2, 0)   // NUL
buf.write_u8(3, 67)  // C
buf.write_u8(4, 68)  // D

let s = buf.to_string()
print(s.len())       // 5 -- the string really is five bytes

let fd = tcp_connect("127.0.0.1", 8792)
tcp_write(fd, s)     // ...but only two bytes, "AB", reach the peer
tcp_close(fd)
```

Reads truncate the same way. Treat the whole `tcp_*` and `tls_*` family as a text transport; for genuinely binary protocols, encode the payload (Base64, hex) before it touches a socket.

Listing 24.11: Reading all data from a TCP connection

```
fn read_all(fd: Int) -> String {
    flux result = ""
    loop {
        let chunk = tcp_read(fd)
        if chunk.len() == 0 {
            break  // EOF
        }
        result = result + chunk
    }
    return result
}
```

If you need to read an exact number of bytes, use `tcp_read_bytes`:

Listing 24.12: Reading an exact number of bytes

```
let fd = tcp_connect("example.com", 80)
tcp_write(fd, "GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n")

// Read exactly 1024 bytes (or less if the connection closes)
let data = tcp_read_bytes(fd, 1024)
print("Received " + to_string(data.len()) + " bytes")

tcp_close(fd)
```

24.2.2 Building a TCP Server

To accept incoming connections, use `tcp_listen` and `tcp_accept`:

Listing 24.13: A minimal TCP echo server

```
let server_fd = tcp_listen("0.0.0.0", 9000)
print("Echo server listening on port 9000")

loop {
    let client_fd = tcp_accept(server_fd)
    let peer = tcp_peer_addr(client_fd)
    print("Connection from " + peer)

    // Read what the client sends
    let data = tcp_read(client_fd)

    // Echo it back
    tcp_write(client_fd, data)

    tcp_close(client_fd)
    print("Closed connection from " + peer)
}
```

`tcp_listen` creates a server socket bound to the given address and port, with `SO_REUSEADDR` set so you can restart the server without waiting for the socket to time out. `tcp_accept` blocks until a client connects, then returns a new socket descriptor for that connection. `tcp_peer_addr` returns the remote address as a `"ip:port"` string.

Socket Timeouts

Use `tcp_set_timeout(fd, seconds)` to set read and write timeouts on a socket. This is important for servers—without a timeout, a misbehaving client that never sends data will block your server forever:

```
let client_fd = tcp_accept(server_fd)
tcp_set_timeout(client_fd, 10)  // 10-second timeout
```

24.2.3 TCP Function Reference

Function	Description
`tcp_listen(host, port)`	Bind and listen; returns server socket descriptor
`tcp_accept(server_fd)`	Accept a connection; returns client socket descriptor
`tcp_connect(host, port)`	Connect to a remote server; returns socket descriptor
`tcp_read(fd)`	Read up to 8 KB; returns String (empty on EOF)
`tcp_read_bytes(fd, n)`	Read exactly `n` bytes; returns String
`tcp_write(fd, data)`	Write all data; handles partial writes
`tcp_close(fd)`	Close socket and release tracking
`tcp_peer_addr(fd)`	Get remote `"ip:port"` as String
`tcp_set_timeout(fd, secs)`	Set read/write timeout in seconds

24.3 TLS: Secure Connections

For encrypted communication, Lattice provides TLS functions that mirror the TCP API but add encryption and certificate verification. The TLS layer is built on OpenSSL (or Schannel on Windows) and supports TLS 1.2 and above.

24.3.1 Making a TLS Connection

Listing 24.14: Connecting with TLS

```
// Check if TLS support is available in this build
if !tls_available() {
    print("TLS not available -- build with OpenSSL support")
    return
}

let fd = tls_connect("example.com", 443)

let request = "GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n"
tls_write(fd, request)

flux response = ""
loop {
    let chunk = tls_read(fd)
    if chunk.len() == 0 { break }
    response = response + chunk
}

print(response)
tls_close(fd)
```

`tls_connect` performs a TCP connection, then negotiates a TLS handshake on top of it. It sets the Server Name Indication (SNI) extension and enables hostname verification, so the connection is secure by default. The returned descriptor is the same integer type as TCP descriptors, but you *must* use the `tls_*` functions to read and write—the data must pass through the encryption layer.

Don't Mix TCP and TLS Functions

A descriptor returned by `tls_connect` must be used with `tls_read`, `tls_write`, and `tls_close`. If you call `tcp_read` on a TLS descriptor, you will read encrypted gibberish. If you call `tcp_close` instead of `tls_close`, the TLS session will not be properly shut down.

24.3.2 TLS Function Reference

Function	Description
`tls_connect(host, port)`	TCP connect + TLS handshake + cert verification
`tls_read(fd)`	Read up to 8 KB of decrypted data
`tls_read_bytes(fd, n)`	Read exactly n decrypted bytes
`tls_write(fd, data)`	Encrypt and write all data
`tls_close(fd)`	TLS shutdown + close socket
`tls_available()`	Returns `true` if TLS support was compiled in

When to Use TLS Directly

In practice, you rarely need the TLS functions directly. `http_get` and friends handle TLS automatically for `https://` URLs. The TLS layer is useful when you need to communicate over encrypted sockets with non-HTTP protocols: database connections, message queues, custom binary protocols, or connecting to services like Redis over TLS.

24.4 Building a Simple HTTP Server

Now that we understand both the HTTP client and the TCP layer, let's build something on the server side. Lattice ships with two approaches: you can build a server from scratch using raw TCP (instructive), or use the `http_server` library for a more productive, framework-style experience.

24.4.1 A Server from Scratch

The minimal approach constructs HTTP responses as raw strings:

Listing 24.15: A minimal HTTP server from raw TCP

```
fn build_response(status: Int, status_text: String,
                  content_type: String, body: String) -> String {
    let header = "HTTP/1.1 " + to_string(status) + " " + status_text + "\r\n"
        + "Content-Type: " + content_type + "\r\n"
        + "Content-Length: " + to_string(body.len()) + "\r\n"
        + "Connection: close\r\n"
        + "\r\n"
    return header + body
}

let greeting = Map::new()
greeting["message"] = "Hello!"

let server = tcp_listen("0.0.0.0", 8080)
print("Listening on http://localhost:8080")

loop {
    let conn = tcp_accept(server)
    let raw = tcp_read(conn)

    if raw.len() > 0 {
        // Parse the request line: "GET /path HTTP/1.1"
        let lines = raw.split("\r\n")
        let parts = lines[0].split(" ")
        let method = parts[0]
        let path = if parts.len() > 1 { parts[1] } else { "/" }

        print(method + " " + path)

        let response = if path == "/" {
            build_response(200, "OK", "text/html",
                "<h1>Hello from Lattice!</h1>")
        } else if path == "/json" {
            build_response(200, "OK", "application/json",
                json_stringify(greeting))
        } else {
            build_response(404, "Not Found", "text/plain",
                "404 Not Found: " + path)
        }

        tcp_write(conn, response)
    }
    tcp_close(conn)
}
```

This works, but it has obvious limitations: no routing, no middleware, no cookie handling, and no request body parsing. The `http_server` library handles all of that.

24.4.2 The http_server Library

Import the library and create an application:

Listing 24.16: Creating an HTTP server with the http_server library

```
import "lib/http_server" as http

let app = http.new()
```

Registering Routes

Routes are registered by HTTP method:

Listing 24.17: Registering route handlers

```
let created = Map::new()
created["status"] = "created"

let deleted = Map::new()
deleted["status"] = "deleted"

app.get("/", |req, res| {
    return http.response(200, "Hello, World!")
})

app.get("/about", |req, res| {
    return http.html(200, "<h1>About</h1><p>Built with Lattice.</p>")
})

app.post("/api/items", |req, res| {
    let body = req.get("body")
    print("Received: " + body)
    return http.json(201, created)
})

app.delete("/api/items", |req, res| {
    return http.json(200, deleted)
})
```

The app object is itself a Map, which raises the question we hit with the `http` module earlier: does `app.get` collide with `Map.get`? It does not. Route registration passes two arguments, and the built-in `Map.get` takes one, so the call falls through to the closure the library stored under that key. Only a single-argument `.get(x)` is ambiguous.

Beyond the four shown, the library also registers `put`, `patch`, `options`, and `all` (which binds one handler to every method).

Route handlers receive a request Map and a response context Map. They must return a response Map, which you build using the helper functions:

Helper	Description
`http.response(status, body)`	Plain text response
`http.json(status, data)`	JSON response (auto-stringifies the data)
`http.html(status, body)`	HTML response
`http.redirect(url)`	302 redirect (optional status parameter)
`http.stream(status, chunks)`	Chunked transfer encoding response
`http.sse_stream(events)`	Server-Sent Events stream

The Request Map

The request Map passed to handlers contains:

Key	Type	Description
`"method"`	`String`	HTTP method ("GET", "POST", etc.)
`"path"`	`String`	Request path without query string
`"raw_path"`	`String`	Request path *with* the query string
`"query"`	`Map`	Parsed query parameters
`"headers"`	`Map`	Request headers
`"body"`	`String`	Request body
`"cookies"`	`Map`	Parsed cookies
`"peer"`	`String`	Remote address (`"ip:port"`)

Listing 24.18: Working with query parameters and cookies

```
app.get("/search", |req, res| {
    let query = req.get("query")
    let term = if query.has("q") { query.get("q") } else { "" }

    flux payload = Map::new()
    payload["results"] = []
    payload["query"] = term
    return http.json(200, payload)
})

app.get("/profile", |req, res| {
    let cookies = req.get("cookies")
    let session = if cookies.has("session") {
        cookies.get("session")
    } else {
        "anonymous"
    }

    flux payload = Map::new()
    payload["user"] = session

    let response = http.json(200, payload)
    return http.set_cookie(response, "visited", "true", http.cookie_opts())
})
```

Middleware

Middleware functions run before route handlers. Each middleware receives the request, response context, and a `next` function to call the rest of the pipeline:

Listing 24.19: Adding middleware to a server

```
import "lib/http_server" as http

let app = http.new()

// Built-in logging middleware
app.use(http.logger())

// Built-in CORS middleware
app.use(http.cors())

// Built-in security headers
app.use(http.security_headers())

// Built-in JSON body parser
app.use(http.body_parser())

// Custom middleware: authentication
app.use(|req, res, next| {
    let headers = req.get("headers")
    if headers.has("Authorization") {
        req.set("authenticated", true)
    } else {
        req.set("authenticated", false)
    }
    return next(req, res)
})

let denied = Map::new()
denied["error"] = "Unauthorized"

let welcome = Map::new()
welcome["message"] = "Welcome, admin"

app.get("/admin", |req, res| {
    if !req.get("authenticated") {
        return http.json(401, denied)
    }
    return http.json(200, welcome)
})

app.listen(8080)
```

Run that and the pipeline is visible in the response itself:

Listing 24.20: Two requests against the middleware pipeline

```
$ curl -i http://127.0.0.1:8080/admin
HTTP/1.1 401 Unauthorized
Content-Length: 24
Connection: close
Content-Type: application/json
X-Frame-Options: DENY
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Referrer-Policy: strict-origin-when-cross-origin
Access-Control-Allow-Origin: *

{"error":"Unauthorized"}

$ curl -H 'Authorization: Bearer x' http://127.0.0.1:8080/admin
{"message":"Welcome, admin"}
```

The middleware pipeline executes in registration order. The `logger` middleware prints timing information for each request. The `cors` middleware handles preflight OPTIONS requests and sets the appropriate `Access-Control-*` headers. The `security_headers` middleware adds `X-Content-Type-Options`, `X-Frame-Options`, and similar defensive headers.

Custom CORS Configuration

The `cors()` middleware accepts an options Map:

```
let cors_config = http.cors_opts()
cors_config["origin"] = "https://mysite.com"
cors_config["credentials"] = true
app.use(http.cors(cors_config))
```

Starting the Server

Listing 24.21: Starting the server

```
app.listen(8080)
// "Lattice HTTP server listening on http://localhost:8080"
```

The `listen` call blocks forever, accepting connections in a loop. Each connection is handled synchronously. For concurrent handling, you could combine this with structured concurrency (Chapter 19), spawning a task for each accepted connection.

24.4.3 A Complete REST API

Let's put it all together with a complete in-memory REST API:

Listing 24.22: A complete REST API server

```
import "lib/http_server" as http

let app = http.new()
app.use(http.logger())
app.use(http.cors())
app.use(http.body_parser())

// In-memory data store
flux items = []
flux next_id = 1

// List all items
app.get("/api/items", |req, res| {
    return http.json(200, items)
})

// Create an item
app.post("/api/items", |req, res| {
    let body = req.get("body")
    let data = json_parse(body)
    data["id"] = next_id
    next_id += 1
    items.push(data)
    return http.json(201, data)
})

// Health check
app.get("/health", |req, res| {
    flux status = Map::new()
    status["status"] = "ok"
    status["items"] = items.len()
    return http.json(200, status)
})

print("Starting API server...")
app.listen(3000)
```

Listing 24.23: Exercising the API

```
$ curl -X POST -d '{"name":"widget"}' http://127.0.0.1:3000/api/items
{"id":1,"name":"widget"}
$ curl -X POST -d '{"name":"gadget"}' http://127.0.0.1:3000/api/items
{"id":2,"name":"gadget"}
$ curl http://127.0.0.1:3000/api/items
[{"id":1,"name":"widget"},{"id":2,"name":"gadget"}]
$ curl http://127.0.0.1:3000/health
{"items":2,"status":"ok"}
```

Why `items.push` Works Now

That `items.push(data)` inside a route handler is doing something older Lattice could not. Until v0.6.4 a mutating builtin method called on a value captured from an enclosing scope operated on a copy and threw the result away, which is why library code from that era—including `http_server` itself—stores routes in pre-allocated parallel arrays and assigns by index. As of v0.6.4 the bytecode backends write the mutation back into the captured cell, so a handler can push onto a shared `flux` array and every later request sees it. The parallel-array idiom you will find inside `lib/http_server.lat` is now history rather than necessity.

24.5 URL Encoding and Decoding

URLs can only contain a limited set of ASCII characters. When you need to include special characters—spaces, ampersands, Unicode text—you must percent-encode them. Lattice provides `url_encode` and `url_decode` for this purpose.

Listing 24.24: URL encoding and decoding

```
let query = "name=Alice Bob&city=New York"
let encoded = url_encode(query)
print(encoded)  // "name%3DAlice%20Bob%26city%3DNew%20York"

let decoded = url_decode(encoded)
print(decoded)  // "name=Alice Bob&city=New York"
```

`url_encode` percent-encodes every character that is not an unreserved character (A–Z, a–z, 0–9, hyphen, underscore, period, tilde). `url_decode` reverses the process, converting `%XX` sequences back to the original bytes and converting `+` to spaces.

Listing 24.25: Building a URL with query parameters

```
fn build_url(base: String, params: Map) -> String {
    let keys = params.keys()
    if keys.len() == 0 {
        return base
    }

    flux parts = []
    for key in keys {
        let encoded_key = url_encode(key)
        let encoded_val = url_encode(params[key])
        parts.push(encoded_key + "=" + encoded_val)
    }

    return base + "?" + parts.join("&")
}

let params = Map::new()
params["q"] = "lattice programming"
params["page"] = "1"
params["lang"] = "en"

let url = build_url("https://search.example.com/api", params)
print(url)
// https://search.example.com/api?lang=en&page=1&q=lattice%20programming
```

Look closely at that output: the parameters come back in a different order than we inserted them. `Map.keys()` reflects the hash table's layout, not insertion order, so a URL built this way is stable for a given set of keys but not predictable from the source. If a signature or a cache key depends on parameter order, sort the keys first.

Module vs. Top-Level Access

`url_encode` and `url_decode` are available both as top-level built-in functions and as members of the built-in `crypto` module—they live there because they are conceptually about encoding, not about cryptography. The module has to be imported under a name before you can reach them:

```
import "crypto" as crypto
print(crypto.url_encode("a b"))  // "a%20b"
```

There is no ambient `crypto` variable; without the `import` you get `undefined variable 'crypto'`. The top-level versions are convenient for scripts; the module versions are better for explicit imports in larger programs.

Exercises

1. **HTTP Health Checker.** Write a program that takes a list of URLs from a configuration file, makes an HTTP GET request to each one, and reports whether the service is up (2xx status), degraded (5xx), or unreachable (network error). Use `try`/`catch` to handle connection failures gracefully.

2. **Web Scraper.** Write a function that fetches a web page with `http_get`, extracts all URLs from `href="..."` attributes using string operations (or the regex module from Chapter 25), and returns them as an array. Bonus: follow the links one level deep and report the status of each.

3. **TCP Chat Server.** Extend the echo server from Section 24.2.2 into a simple chat server. When a client connects, ask for their name, then broadcast each message to all connected clients. You will need to use structured concurrency (Chapter 19) to handle multiple clients simultaneously.

4. **REST API with Persistence.** Extend the REST API from Section 24.4.3 to persist its data to a JSON file. Load the file on startup and save after each modification. Use the safe-write pattern from Section 22.4.1 in the filesystem chapter.

5. **HTTP File Server.** Build a server that serves static files from a directory. Map the URL path to a file path, read the file, determine the content type from the extension, and send it back. Handle directory traversal attacks by checking that the resolved path stays within the root directory (use `realpath`).

What's Next

We have now covered the major I/O systems in Lattice: filesystems, data formats, and networking. In the final chapter of this Part, we will survey the remaining standard library modules—time and

date handling, cryptographic functions, UUIDs, operating system interaction, shell-free subprocess execution with `exec_argv`, progress reporting, mathematics, and regular expressions—giving you the complete toolkit for building real-world applications.

Chapter 25

Time, Crypto, OS, and Everything Else

This chapter is a grand tour of the remaining standard library modules—the functions that do not fit neatly into files, formats, or networking, but that you will reach for constantly in real programs. We will cover timestamps and date arithmetic, cryptographic hashing and encoding, UUID generation, operating-system interaction, shell-free subprocess execution, progress reporting, mathematics, and regular expressions. Each section is self-contained, so feel free to jump to whichever module you need right now and come back to the others later.

25.1 The time Module

Time in Lattice is represented as a single integer: the number of milliseconds since the Unix epoch (January 1, 1970, 00:00:00 UTC). This is the same convention used by JavaScript's `Date.now()`, making it straightforward to exchange timestamps with web APIs.

25.1.1 Getting the Current Time

Listing 25.1: Getting the current timestamp

```
let now = time()
print(now)  // e.g., 1703456789012

// Measure execution time
let start = time()
// ... some computation ...
let elapsed = time() - start
print("Took " + to_string(elapsed) + " ms")
```

`time()` returns the current wall-clock time as an `Int` in milliseconds. Under the hood, it calls `clock_gettime` with `CLOCK_REALTIME` (see `src/time_ops.c`).

25.1.2 Sleeping

Listing 25.2: Pausing execution

```
print("Starting...")
sleep(1000)  // Sleep for 1 second
print("Done!")

// Short delay for polling
sleep(100)  // 100 milliseconds
```

`sleep` takes a duration in milliseconds. It uses `nanosleep` for precise timing and handles `EINTR` gracefully.

sleep Is Not Available in WASM

In the browser (Emscripten) build, `sleep` raises a runtime error because it would block the single-threaded event loop. If you need time-delayed behavior in WASM, use callbacks or an async pattern instead.

25.1.3 Formatting Timestamps

Listing 25.3: Formatting a timestamp as a human-readable string

```
let now = time()

let readable = time_format(now, "%Y-%m-%d %H:%M:%S")
print(readable)  // "2024-12-25 14:30:45"

let date_only = time_format(now, "%B %d, %Y")
print(date_only)  // "December 25, 2024"

let time_only = time_format(now, "%I:%M %p")
print(time_only)  // "02:30 PM"
```

`time_format` takes a timestamp (in epoch milliseconds) and a format string using the standard `strftime` specifiers:

Specifier	Meaning	Specifier	Meaning
%Y	4-digit year	%H	Hour (00–23)
%m	Month (01–12)	%I	Hour (01–12)
%d	Day (01–31)	%M	Minute (00–59)
%B	Full month name	%S	Second (00–59)
%b	Abbreviated month	%p	AM/PM
%A	Full weekday name	%Z	Timezone abbreviation
%a	Abbreviated weekday	%%	Literal %

25.1.4 Parsing Timestamps

Listing 25.4: Parsing a date string into a timestamp

```
let timestamp = time_parse("2024-12-25 14:30:00", "%Y-%m-%d %H:%M:%S")
print(timestamp)  // epoch milliseconds

let formatted = time_format(timestamp, "%B %d, %Y")
print(formatted)  // "December 25, 2024"
```

`time_parse` is the inverse of `time_format`. It takes a date string and a format pattern, and returns the corresponding epoch timestamp in milliseconds. The format specifiers are the same as for `time_format`.

Local Time

Both `time_format` and `time_parse` work in the system's local timezone by default. The underlying C functions (`localtime_r` and `mktime`) handle timezone conversions and daylight saving time adjustments automatically.

25.1.5 Extracting Date Components

For quick access to individual components without constructing a format string, Lattice provides dedicated extraction functions:

Listing 25.5: Extracting date and time components

```
let now = time()

print(time_year(now))     // 2024
print(time_month(now))    // 12 (1-based)
print(time_day(now))      // 25
print(time_hour(now))     // 14
print(time_minute(now))   // 30
print(time_second(now))   // 45
print(time_weekday(now))  // 3 (0=Sunday, 1=Monday, ..., 6=Saturday)
```

25.1.6 Date Arithmetic

Since timestamps are just integers in milliseconds, date arithmetic is straightforward:

Listing 25.6: Date arithmetic with timestamps

```
let now = time()

// Add 7 days
let next_week = time_add(now, 7 * 24 * 60 * 60 * 1000)
print("Next week: " + time_format(next_week, "%Y-%m-%d"))

// Subtract 30 days
let last_month = time_add(now, -30 * 24 * 60 * 60 * 1000)
print("30 days ago: " + time_format(last_month, "%Y-%m-%d"))

// Useful constants (in milliseconds)
let SECOND = 1000
let MINUTE = 60 * SECOND
let HOUR = 60 * MINUTE
let DAY = 24 * HOUR

let deadline = time_add(now, 3 * HOUR + 30 * MINUTE)
print("Deadline: " + time_format(deadline, "%H:%M:%S"))
```

`time_add` simply adds a delta in milliseconds to a timestamp. You can also use plain addition (`now + delta`), but `time_add` makes the intent clearer.

25.1.7 The datetime Functions

For more structured date/time work—especially when dealing with ISO 8601 strings and UTC conversions—Lattice provides a family of `datetime_*` functions. These work with epoch *seconds* (not milliseconds) and return Maps with structured fields.

Listing 25.7: Working with ISO 8601 dates

```
// Get current datetime as a Map
let now = datetime_now()
print(now)
// {"month": 8, "tz_offset": -18000, "minute": 33, "year": 2026,
//  "day": 4, "second": 57, "hour": 23}

// Parse an ISO 8601 string
let dt = datetime_from_iso("2024-12-25T14:30:00Z")
print(dt)
// {"month": 12, "tz_offset": 0, "minute": 30, "year": 2024,
//  "day": 25, "second": 0, "hour": 14}

// Convert back to ISO 8601
let iso_str = datetime_to_iso(dt)
print(iso_str)  // "2024-12-25T14:30:00Z"

// Format with a custom pattern
let formatted = datetime_format(dt, "%B %d, %Y at %H:%M")
print(formatted)  // "December 25, 2024 at 14:30"

// Round-trip through epoch seconds
print(datetime_to_epoch(dt))  // 1735137000 (seconds, not milliseconds)
```

A datetime Map carries seven fields: `year`, `month`, `day`, `hour`, `minute`, `second`, and `tz_offset` (the offset from UTC in seconds—zero for a value parsed from a `Z`-suffixed ISO string).

Durations Are Maps Too

The one thing you cannot do is hand `datetime_add_duration` a plain number of seconds. Durations are their own Map type. Build one with `duration(hours, minutes, seconds, millis)`, or from a raw count with `duration_from_seconds` and `duration_from_millis`:

Listing 25.8: Duration arithmetic with datetime

```
let start = datetime_from_iso("2024-01-15T09:00:00Z")

let ninety = duration(1, 30, 0, 0)   // 1h 30m 0s 0ms
print(ninety)
// {"hours": 1, "total_ms": 5400000, "millis": 0, "seconds": 0, "minutes": 30}

let meeting_end = datetime_add_duration(start, ninety)
print(datetime_to_iso(meeting_end))  // "2024-01-15T10:30:00Z"

// datetime_sub returns a duration Map, not a number
let finish = datetime_from_iso("2024-01-15T10:30:00Z")
let diff = datetime_sub(finish, start)
print(duration_to_string(diff))  // "1h 30m 0s"
print(duration_minutes(diff))    // 30
```

`datetime_add_duration` Wants a Duration

Passing an Int where a duration Map belongs fails loudly rather than guessing at units:

```
datetime_add_duration(start, 90 * 60)
// vm error: datetime_add_duration: expected (dt: Map, dur: Map)
```

The `--tree-walk` wording is `datetime_add_duration() expects (Map dt, Map dur)`; the core complaint is the same on every backend. Reach for `duration_from_seconds(90 * 60)` when you have a raw count.

A duration Map exposes `hours`, `minutes`, `seconds`, `millis`, and a `total_ms` field, and the accessor functions `duration_hours`, `duration_minutes`, `duration_seconds`, and `duration_millis` read them back. `duration_add` and `duration_sub` combine two of them.

The `datetime_to_utc` and `datetime_to_local` functions convert between UTC and local time.

`timezone_offset()` returns the local offset from UTC in seconds. In the US Eastern zone used for the transcripts in this chapter, that is `-18000`.

Which Time API to Use?

For simple timing and logging, the `time()` / `time_format()` functions are sufficient. For applications that exchange dates with APIs, store them in databases, or perform calendar arithmetic across timezones, use the `datetime_*` family with ISO 8601 strings.

25.2 The crypto Module

Lattice provides cryptographic primitives for hashing, message authentication, encoding, and random number generation. When built with OpenSSL support (`LATTICE_HAS_TLS`), these functions use OpenSSL's optimized implementations. Without OpenSSL, Lattice falls back to pure-C implementations that produce identical results—just more slowly.

25.2.1 Hashing: sha256, sha512, md5

Listing 25.9: Computing cryptographic hashes

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

let hash = sha256(message)
print(hash)  // 64-character hex string
// 63b7dec05f181a28f2c8307f4fbe1149be7f911b8af9deb3af5a8073fa967bc4

let long_hash = sha512(message)
print(long_hash.len())  // 128

let legacy_hash = md5(message)
print(legacy_hash)  // 50704ca538d2ef2b87faf60b94be120f
```

All hash functions take a `String` and return a lowercase hexadecimal string. `sha256` produces a 256-bit (64 hex character) digest. `sha512` produces a 512-bit (128 hex character) digest. `md5` produces a 128-bit (32 hex character) digest.

MD5 Is Not Secure

MD5 is cryptographically broken and should never be used for security purposes (password hashing, digital signatures, integrity verification of untrusted data). It is included for backward compatibility with legacy systems. Use `sha256` or `sha512` for new applications.

Listing 25.10: File integrity checking with sha256

```
fn file_checksum(path: String) -> String {
    let contents = read_file(path)
    return sha256(contents)
}

let expected = "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
let actual = file_checksum("important_data.txt")

if actual == expected {
    print("File integrity verified")
} else {
    print("WARNING: File has been modified!")
}
```

25.2.2 HMAC: hmac_sha256

HMAC (Hash-based Message Authentication Code) combines a secret key with a hash function to verify both the integrity and authenticity of a message:

Listing 25.11: Computing an HMAC

```
let secret_key = "my-secret-key"
let message = "user=alice&amount=100"

let mac = hmac_sha256(secret_key, message)
print(mac)
// 78290c56c56e13b902e30884fef390107da28c547896e73d68674571251cd7f6

// Verify by recomputing
let expected_mac = hmac_sha256(secret_key, message)
if mac == expected_mac {
    print("Message is authentic")
}
```

`hmac_sha256` takes a key and a message, both as strings, and returns a hex-encoded MAC. This is the standard way to sign API requests and verify webhook payloads.

25.2.3 Base64 Encoding and Decoding

Base64 converts arbitrary binary data into a text-safe ASCII representation:

Listing 25.12: Base64 encoding and decoding

```
let original = "Hello, World!"

let encoded = base64_encode(original)
print(encoded)  // "SGVsbG8sIFdvcmxkIQ=="

let decoded = base64_decode(encoded)
print(decoded)  // "Hello, World!"
```

The Decoder Is Strict

As of v0.6.4, `base64_decode` validates its input instead of skipping over garbage. Padding in the wrong place, stray characters, a bad alphabet—all of it raises a catchable error:

```
try { print(base64_decode("a===")) }
catch e { print(e.message) }
// base64_decode: invalid character in input
```

Identical on all three backends. Code that used to lean on the old forgiving behaviour—feeding it URL-safe Base64, or concatenated chunks with interior padding—now needs to normalize the input first.

Base64 is commonly used for embedding binary data in JSON, passing data in URL parameters, and encoding credentials for HTTP Basic authentication:

Listing 25.13: Base64 credentials for HTTP Basic authentication

```
let username = "admin"
let password = "s3cret"
let credentials = base64_encode(username + ":" + password)
print("Basic " + credentials)  // "Basic YWRtaW46czNjcmV0"
```

Getting That Header onto the Wire

Building the credential string is the straightforward half. Attaching it is where v0.6.4 gets in the way: as Section 24.1.2 explains, the bytecode backends read only the `"body"` key out of an HTTP options Map, so an `"Authorization"` header set this way travels only under `--tree-walk`. Until that is harmonized, authenticate through the URL, the body, or a hand-written request over `tcp_write`.

25.2.4 Random Bytes

For cryptographically secure random data, use `random_bytes`:

Listing 25.14: Generating random bytes

```
let token_bytes = random_bytes(32)
print(token_bytes.to_hex())  // 64-character random hex string

// Generate a random hex token
fn generate_token(length: Int) -> String {
    let bytes = random_bytes(length)
    return bytes.to_hex()
}

let session_token = generate_token(16)
print("Session: " + session_token)
```

`random_bytes` returns a `Buffer` of the specified length, filled with cryptographically secure random bytes. On macOS and BSD, it uses `arc4random_buf`. On Linux, it reads from `/dev/urandom`. On Windows, it uses `RtlGenRandom`. With OpenSSL available, it uses `RAND_bytes`. All of these are suitable for generating session tokens, encryption keys, and nonces.

Cryptographic vs. Pseudorandom

`random_bytes` produces cryptographically secure random data—it is unpredictable and suitable for security-sensitive applications. The `random()` function from the math module (see Section 25.5.5) uses `rand()`, which is *not* cryptographically secure and should only be used for non-security purposes like games and simulations.

25.3 uuid()

Universally Unique Identifiers (UUIDs) are 128-bit identifiers typically displayed as 32 hexadecimal digits in a 8-4-4-4-12 format. Lattice provides a built-in function for generating Version 4 (random) UUIDs:

Listing 25.15: Generating UUIDs

```
let id = uuid()
print(id)  // "550e8400-e29b-41d4-a716-446655440000" (example)

// Generate multiple unique identifiers
flux ids = []
flux i = 0
while i < 5 {
    ids.push(uuid())
    i += 1
}
for id in ids {
    print(id)
}
```

Each call to `uuid()` returns a fresh, unique string. The underlying implementation uses `random_bytes` to generate 16 random bytes, then sets the version bits (0100 for V4) and variant bits (10xx for RFC 4122) before formatting.

Listing 25.16: UUIDs for record identifiers

```
fn create_user(name: String, email: String) -> Map {
    let user = Map::new()
    user["id"] = uuid()
    user["name"] = name
    user["email"] = email
    user["created_at"] = time_format(time(), "%Y-%m-%dT%H:%M:%SZ")
    return user
}

let alice = create_user("Alice", "alice@example.com")
print(json_stringify(alice))
```

25.4 The os Module

The operating system functions let your program interact with its environment: reading and setting environment variables, inspecting command-line arguments, executing external commands, and querying system information.

25.4.1 Environment Variables: env, env_set, env_keys

Listing 25.17: Working with environment variables

```
// Read an environment variable
let home = env("HOME")
print("Home directory: " + home)

let path = env("PATH")
print("PATH: " + path)

// Check for a variable -- an unset name yields Unit, not nil
let debug = env("DEBUG")
if typeof(debug) == "Unit" {
    print("Debug mode is off")
} else {
    print("Debug mode is on: " + debug)
}

// Set an environment variable
env_set("APP_MODE", "production")
print(env("APP_MODE"))  // "production"

// List all environment variable names
let keys = env_keys()
print("Environment has " + to_string(keys.len()) + " variables")
```

`env(name)` returns the value of the named environment variable as a `String`. `env_set` sets a variable for the current process (it does not modify the parent shell's environment). `env_keys` returns an array of all environment variable names.

An Unset Variable Is Unit, Not Nil

The declared signature is `env(name: String) -> String|Unit`. That second alternative matters: a missing variable comes back as `Unit`, and `Unit == nil` is `false`. Guard with `typeof`:

```
let missing = env("DEFINITELY_NOT_SET")
print(missing == nil)        // false  -- the trap
print(typeof(missing))       // "Unit" -- the truth
```

This is the opposite of `input()`, which as of v0.6.4 returns a genuine `nil` at end of file. The two are routinely conflated; the compiler will not catch it for you.

25.4.2 Command-Line Arguments: args()

Listing 25.18: Accessing command-line arguments

```
// Run: clat script.lat --verbose output.txt

let arguments = args()
print(arguments)  // [script.lat, --verbose, output.txt]

// Simple argument parsing
flux verbose = false
flux output_path = "default.txt"

flux i = 0
while i < arguments.len() {
    if arguments[i] == "--verbose" {
        verbose = true
    } else if !arguments[i].starts_with("--") {
        output_path = arguments[i]
    }
    i += 1
}

if verbose {
    print("Writing to: " + output_path)
}
```

`args()` returns the command-line arguments as an array of strings. The first element is typically the script name.

25.4.3 System Information: platform, cwd, hostname, pid

Listing 25.19: Querying system information

```
print("Platform: " + platform())     // "macos", "linux", "windows", or "wasm"
print("Working dir: " + cwd())       // "/home/user/project"
print("Hostname: " + hostname())     // "workstation"
print("Process ID: " + to_string(pid()))  // 12345
```

`platform()` returns a short string identifying the operating system. This is useful for writing cross-platform scripts that need to adjust their behavior:

Listing 25.20: Platform-specific behavior

```
let separator = if platform() == "windows" { "\\" } else { "/" }
let config_dir = if platform() == "macos" {
    env("HOME") + "/Library/Application Support/myapp"
} else if platform() == "linux" {
    env("HOME") + "/.config/myapp"
} else {
    env("APPDATA") + "\\myapp"
}

if !file_exists(config_dir) {
    mkdir(config_dir)
}
```

25.4.4 Executing External Commands: exec and shell

Listing 25.21: Running external commands with exec

```
// exec runs a command and returns its stdout as a string
let output = exec("echo 'Hello from the shell'")
print(output)  // "Hello from the shell\n"

// Capture the output of a system command
let git_status = exec("git status --short")
print(git_status)
```

`exec` runs a command via popen, captures its standard output, and returns it as a string. If the command exits with a non-zero status, `exec` raises a runtime error:

Listing 25.22: A non-zero exit from exec is an error

```
try { exec("exit 3") }
catch e { print(e.message) }
// exec: command failed with exit code 3
```

For more control—including access to both stdout, stderr, and the exit code—use `shell`:

Listing 25.23: Running commands with shell for full control

```
let result = shell("ls -la /nonexistent")

print("Exit code: " + to_string(result["exit_code"]))
print("Stdout: " + result["stdout"])
print("Stderr: " + result["stderr"])

// Check if a command succeeded
if result["exit_code"] == 0 {
    print("Success!")
} else {
    print("Failed: " + result["stderr"])
}
```

`shell` returns a Map with three keys: `"exit_code"` (an `Int`), `"stdout"` (a `String`), and `"stderr"` (a `String`). Unlike `exec`, it does *not* raise an error on non-zero exit codes, giving you full control over error handling.

Shell Injection

Both `exec` and `shell` pass the command string directly to the system shell (`/bin/sh` on Unix, `cmd.exe` on Windows). **Never** construct command strings from untrusted user input without sanitization—this is a classic shell injection vulnerability:

```
// DANGEROUS: user_input could contain "; rm -rf /"
let result = exec("echo " + user_input)
```

The durable fix is not better escaping. It is to stop involving a shell at all—which is exactly what the next section is about.

25.4.5 exec_argv: Running a Program Without a Shell

Every shell-injection bug in history starts the same way: a program glued a command line together out of strings and handed the result to an interpreter that reads punctuation as instructions. `exec_argv`, added in the v0.6.3 window, removes the interpreter from the loop. You name the program and you pass its arguments as an array, and the kernel receives exactly that vector—no word splitting, no globbing, no metacharacters.

Listing 25.24: Running a program with exec_argv

```
let r = exec_argv("printf", ["%s", "hi"], nil)

print(r["stdout"])                // "hi"
print(r["exit_code"])             // 0
print(r["stderr"] == "")          // true
```

The signature is `exec_argv(program, args, stdin, options = nil)`:

Parameter	Type	Meaning
`program`	`String`	Program to run; resolved through `PATH`
`args`	`Array`	Arguments *excluding* `argv[0]`
`stdin`	`String|Nil`	Data to feed the child; `nil` closes its input immediately
`options`	`Map|Nil`	Optional execution limits (below)

The return value is a Map with exactly three keys—`"stdout"`, `"stderr"`, and `"exit_code"`.

A Non-Zero Exit Is Data

Unlike exec, a failing child is not an error. You get its exit code and its diagnostics and you decide what they mean:

Listing 25.25: Failure is a value, not an exception

```
let bad = exec_argv("sh", ["-c", "echo oops 1>&2; exit 3"], nil)

print(bad["exit_code"])        // 3
print(bad["stderr"].trim())    // "oops"
```

stdin Without a Pipeline

The third argument is written to the child's standard input and the pipe is then closed, so filters work without any shell plumbing:

Listing 25.26: Feeding a child process

```
let counted = exec_argv("wc", ["-c"], "abcde")
print(counted["stdout"].trim())  // "5"
```

Why This Is Safe

Here is the payload that would end a career if it reached `/bin/sh`:

Listing 25.27: Metacharacters stay ordinary characters

```
let literal = exec_argv("printf", ["%s", "; rm -rf /"], nil)
print(literal["stdout"])  // "; rm -rf /"
```

The semicolon is a semicolon. There is no shell to read it as a command separator, because `exec_argv` goes straight to `posix_spawnp` (`CreateProcess` on Windows) with the argument vector you supplied. See `src/process_ops.c` if you want to watch it happen.

Bounding the Child

The optional fourth argument is a Map of limits. Only three keys are recognized—anything else is refused rather than ignored:

Option	Effect
`"timeout_ms"`	Kill the child (and its descendants) after this many milliseconds
`"max_stdout_bytes"`	Abort once captured stdout exceeds this many bytes
`"max_stderr_bytes"`	Same, for stderr

Listing 25.28: Limits raise catchable errors

```
let opts = Map::new()
opts["timeout_ms"] = 200
try { exec_argv("sleep", ["5"], nil, opts) }
catch e { print(e.message) }
// exec_argv: timed out after 200 ms

let lim = Map::new()
lim["max_stdout_bytes"] = 4
try { exec_argv("printf", ["%s", "abcdefgh"], nil, lim) }
catch e { print(e.message) }
// exec_argv: stdout exceeded max_stdout_bytes (4 bytes)

let typo = Map::new()
typo["retries"] = 3
try { exec_argv("printf", ["hi"], nil, typo) }
catch e { print(e.message) }
// exec_argv: unknown option 'retries'

try { exec_argv("definitely-not-a-program", [], nil) }
catch e { print(e.message) }
// exec_argv: failed to spawn 'definitely-not-a-program':
//   No such file or directory
```

Every message above is byte-identical on the default stack VM, `--tree-walk`, and `--regvm`. When any limit is configured, the child is spawned into its own process group, so a timeout takes its descendants with it rather than leaving orphans behind; and a child that closes its input early cannot take the interpreter down with `SIGPIPE`.

Binary Output Survives

`exec_argv` captures the child's output with an explicit length, so NUL bytes come through intact. A child that prints `41 00 42` gives you a three-byte string, and `Buffer::from_string(r["stdout"]).len()` agrees. That is a real difference from the socket and HTTP layers of Chapter 24, which stop at the first NUL.

Not Available in the Browser

The WebAssembly build has no process model. Calling `exec_argv` there raises the catchable error `exec_argv: not available in browser` rather than failing at load time, so a program can probe for it and fall back.

Choosing Between exec, shell, and exec_argv

Reach for `exec_argv` by default. Use `shell` only when you genuinely want shell features—pipelines, redirection, glob expansion—on a command string you wrote yourself. Use `exec` when you want stdout and want a non-zero exit to abort. Any command line that contains a value you did not write belongs in `exec_argv`.

Seen in the Wild: the Ballistics Lab

The largest example program shipped with Lattice, `examples/ballistics_lab/`, is built around this function. The Lab is a stateful domain REPL—units, immutable domain values, tables, JSON/CSV export, transactional sessions—written entirely in Lattice, with no native extension anywhere in it. Every solve is a JSON request handed to an external ballistics executable through `exec_argv`, wrapped in per-call `timeout_ms`, `max_stdout_bytes`, and `max_stderr_bytes` limits, with the response validated field by field before it is trusted. If you want to see what a serious `exec_argv` integration looks like, read `examples/ballistics_lab/process_backend.lat`.

25.5 The math Module

Lattice's math functions are registered as top-level built-ins. They accept both `Int` and `Float` arguments (promoting to `Float` when needed) and cover the standard mathematical operations.

25.5.1 Rounding and Absolute Value

Listing 25.29: Rounding functions

```
print(abs(-42))       // 42
print(abs(-3.14))     // 3.14

print(floor(3.7))     // 3
print(floor(-2.3))    // -3

print(ceil(3.2))      // 4
print(ceil(-2.7))     // -2

print(round(3.5))     // 4
print(round(2.4))     // 2
```

`floor`, `ceil`, and `round` all return `Int` values (the C `floor`/`ceil`/`round` cast to `int64_t`). When given an `Int`, they return it unchanged.

25.5.2 Powers, Roots, and Logarithms

Listing 25.30: Powers and roots

```
print(pow(2, 10))     // 1024 (Int)
print(pow(2, -1))     // 0.5 (Float, because negative exponent)
print(sqrt(144))      // 12

print(log(math_e()))   // 1 (natural log)
print(log2(1024))      // 10
print(log10(1000))     // 3

print(exp(1))          // 2.71828 (e^1)
```

`pow` is smart about types: when both arguments are non-negative integers, it returns an `Int` (with overflow detection that falls back to `Float`). `sqrt` always returns a `Float` and raises a domain error for negative inputs: `sqrt() domain error: negative input`.

Floats Print Without Their Tails

`print` formats a Float to six significant digits and drops a trailing `.0`, which is why `sqrt(144)` shows as `12` rather than `12.0` and `exp(1)` stops at `2.71828`. The value is still a full double—only the display is abbreviated. `typeof(sqrt(144))` answers `"Float"`, and `to_string` and `format` use the same six-digit rendering, so if you need more precision in output you will have to build the digits yourself.

25.5.3 Min, Max, Clamp, and Lerp

Listing 25.31: Clamping and interpolation

```
print(min(3, 7))     // 3
print(max(3, 7))     // 7

// Clamp a value to a range
let health = clamp(120, 0, 100)
print(health)  // 100

let brightness = clamp(-10, 0, 255)
print(brightness)  // 0

// Linear interpolation
let midpoint = lerp(0.0, 100.0, 0.5)
print(midpoint)  // 50

let quarter = lerp(10.0, 20.0, 0.25)
print(quarter)  // 12.5
```

`clamp(value, low, high)` constrains a value to the range [*low*, *high*]. `lerp(a, b, t)` computes the linear interpolation $a + (b - a) \times t$. Both are essential for game development, animation, and signal processing.

25.5.4 Trigonometry

Listing 25.32: Trigonometric functions

```
let pi = math_pi()

print(sin(0))           // 0
print(sin(pi / 2))      // 1
print(cos(0))           // 1
print(cos(pi))          // -1
print(tan(pi / 4))      // 1

// Inverse trig
print(asin(1.0))        // 1.5708   (pi/2)
print(acos(0.0))        // 1.5708   (pi/2)
print(atan(1.0))        // 0.785398 (pi/4)

// atan2 for angle from coordinates
let angle = atan2(1.0, 1.0)
print(angle)  // 0.785398 (pi/4, i.e., 45 degrees)
```

All trigonometric functions work in radians. `math_pi()` returns π and `math_e()` returns Euler's number e, both as `Float` constants.

Hyperbolic functions are also available: `sinh(1.0)` is `1.1752`, `cosh(1.0)` is `1.54308`, and `tanh(1.0)` is `0.761594`.

25.5.5 Random Numbers

Listing 25.33: Generating random numbers

```
// Random float in [0, 1)
let r = random()
print(r)  // e.g., 0.7234...

// Random integer in a range [low, high] (inclusive)
let dice = random_int(1, 6)
print("You rolled: " + to_string(dice))

// Shuffle an array using random
fn shuffle(arr: Array) -> Array {
    flux result = arr
    flux i = result.len() - 1
    while i > 0 {
        let j = random_int(0, i)
        let temp = result[i]
        result[i] = result[j]
        result[j] = temp
        i -= 1
    }
    return result
}

let deck = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let shuffled = shuffle(deck)
print(shuffled)
```

`random()` returns a `Float` in the range $[0, 1)$. `random_int(low, high)` returns an `Int` in the inclusive range $[\text{low}, \text{high}]$.

25.5.6 Other Math Functions

Listing 25.34: Additional math functions

```
// Sign function
print(sign(-42))   // -1
print(sign(0))     // 0
print(sign(7.5))   // 1

// Greatest common divisor and least common multiple
print(gcd(12, 8))   // 4
print(lcm(4, 6))    // 12

// Special value detection
print(is_nan(0.0 / 0.0))   // true
print(is_inf(1.0 / 0.0))   // true
print(is_nan(42.0))        // false
```

25.5.7 Where the Numbers Stop

Producing a NaN or an infinity is fine—`0.0 / 0.0` and `1.0 / 0.0` both evaluate happily, as the listing above shows. *Converting* one to an integer is not. Since vo.6.4 every float-to-int crossing is guarded, so a NaN can no longer wander into an array index and turn into a silent, catastrophic zero:

Listing 25.35: Float-to-Int conversions are guarded

```
let nan = 0.0 / 0.0
let inf = 1.0 / 0.0

try { to_int(nan) } catch e { print(e.message) }
// to_int(): Float value out of Int range

try { floor(nan) } catch e { print(e.message) }
// floor(): value out of Int range

try { ceil(inf) }  catch e { print(e.message) }
// ceil(): value out of Int range

try { round(nan) } catch e { print(e.message) }
// round(): value out of Int range
```

All four are catchable, and all four read identically on every backend.

Integer *literals* are checked even earlier. A literal that will not fit in a signed 64-bit integer is a compile-time error rather than a silently saturated or wrapped value:

Listing 25.36: An out-of-range literal never runs

```
$ clat big.lat
error: 1:7: integer literal out of range
```

Other Hard Limits Worth Knowing

Two more ceilings live in the parser, and both report themselves clearly rather than blowing the C stack:

- Expression nesting is capped at 1000. Exceed it and you get `error: expression nesting too deep`. The same `PARSER_MAX_DEPTH` now covers deeply nested patterns and type expressions.
- String-interpolation nesting is capped at 200, reported with a position: `error: 1:607: string interpolation nested too deep`.

You will not meet either writing ordinary code—they exist so that machine-generated or hostile input fails cleanly. Both hold on Windows too, where the interpreter reserves a 16 MB stack precisely so the limits fire before the platform does.

25.6 The regex Module

Regular expressions let you search, match, and transform text using patterns. Lattice uses POSIX Extended Regular Expressions (ERE) on Unix systems.

25.6.1 Matching: regex_match

Listing 25.37: Testing if a string matches a pattern

```
let email = "alice@example.com"
let pattern = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$"

if regex_match(pattern, email) {
    print("Valid email format")
} else {
    print("Invalid email format")
}
```

`regex_match` returns `true` if the pattern matches anywhere in the string. Use `^` and `$` anchors to require a full-string match.

Listing 25.38: Case-insensitive matching with flags

```
// The third argument is a flags string
let is_greeting = regex_match("^hello", "Hello World", "i")
print(is_greeting)  // true
```

The optional flags string supports:

Flag	Meaning
`i`	Case-insensitive matching (`REG_ICASE`)
`m`	Newline-sensitive matching (`REG_NEWLINE`)

Without `m`, a Dot Eats Your Newlines

POSIX ERE without `REG_NEWLINE` treats the subject as one long string, so `.` matches a line feed like any other character and a greedy `.*` will happily swallow an entire file. If you are matching line-oriented text—logs, configuration, anything you split on newlines in your head—pass the `m` flag. We will watch this bite in Section 25.6.2.

25.6.2 Finding All Matches: regex_find_all

Listing 25.39: Finding all pattern matches in a string

```
let text = "Call 555-1234 or 555-5678 for info"
let phone_pattern = "[0-9]{3}-[0-9]{4}"

let numbers = regex_find_all(phone_pattern, text)
for number in numbers {
    print("Found: " + number)
}
// Found: 555-1234
// Found: 555-5678
```

`regex_find_all` returns an array of all non-overlapping matches. If no matches are found, it returns an empty array.

Now the trap the flag table warned about. Here is the same search run twice over a multi-line subject—once without `m`, once with it:

Listing 25.40: Extracting data from structured text

```
let log = """
[2024-01-15] ERROR: Connection timeout
[2024-01-15] INFO: Server started
[2024-01-16] ERROR: Disk space low
[2024-01-16] WARN: Memory usage high
"""

// Without the m flag: one match, swallowing the whole log
let greedy = regex_find_all("\\[.*\\] ERROR: .*", log)
print("Found " + to_string(greedy.len()) + " errors:")
// Found 1 errors:

// With the m flag: line-oriented, as intended
let errors = regex_find_all("\\[.*\\] ERROR: .*", log, "m")
print("Found " + to_string(errors.len()) + " errors:")
for error in errors {
    print("  " + error)
}
// Found 2 errors:
//   [2024-01-15] ERROR: Connection timeout
//   [2024-01-16] ERROR: Disk space low
```

The first pattern is not wrong so much as literal: `.` really does match everything, newlines included, and the greedy `.*` runs from the first `[` to the end of the subject. Adding `m` anchors `.` to a single line and the intent and the behaviour line up again.

25.6.3 Replacing: regex_replace

Listing 25.41: Replacing patterns in text

```
let text = "The quick brown fox jumps over the lazy dog"

// Replace all vowels
let result = regex_replace("[aeiou]", text, "*")
print(result)  // "Th* q**ck br*wn f*x j*mps *v*r th* l*zy d*g"

// Sanitize user input: remove non-alphanumeric characters
let input = "Hello! <script>alert('xss')</script>"
let clean = regex_replace("[^a-zA-Z0-9 ]", input, "")
print(clean)  // "Hello scriptalertxssscript"
```

`regex_replace` replaces *all* non-overlapping matches of the pattern with the replacement string. It returns a new string; the original is unmodified.

Listing 25.42: Normalizing whitespace

```
let messy = "Too   many     spaces    here"
let clean = regex_replace("  +", messy, " ")
print(clean)  // "Too many spaces here"
```

Regex on Windows

The regex functions use POSIX `regex.h`, which is not available in MinGW on Windows. On Windows builds, the regex functions return stub errors. A future version may bundle PCRE2 for cross-platform support. See `src/regex_ops.c` for details.

25.6.4 Regex Size Caps

A pathological pattern applied to a pathological subject is a classic denial-of-service vector, so the regex layer refuses both outright rather than grinding. Patterns are capped at 8192 bytes and subjects at 16 MB (`REGEX_MAX_PATTERN` and `REGEX_MAX_INPUT` in `src/regex_ops.c`):

Listing 25.43: Oversized inputs fail the call

```
let huge_pattern = "a".repeat(9000)
try { regex_match(huge_pattern, "aaa") }
catch e { print(e.message) }
// regex: pattern or input too large

let huge_subject = "a".repeat(17 * 1024 * 1024)
try { regex_match("a", huge_subject) }
catch e { print(e.message) }
// regex: pattern or input too large
```

Both are catchable, and both report the same message on all three backends. If you are scanning something genuinely enormous—a multi-gigabyte log, say—read it in chunks and match each chunk.

25.6.5 Practical Example: Log Parser

Let's combine regex with file I/O and data formats to build a log file parser:

Listing 25.44: A log parser using regex

```
fn parse_log(path: String) -> Array {
    let content = read_file(path)
    let lines = content.split("\n")

    flux entries = []

    for line in lines {
        if line.len() == 0 { continue }

        let matches = regex_find_all(
            "[0-9]{4}-[0-9]{2}-[0-9]{2}", line)
        let levels = regex_find_all("[A-Z]{3,}", line)

        if matches.len() > 0 && levels.len() > 0 {
            let entry = Map::new()
            entry["date"] = matches[0]
            entry["level"] = levels[0]
            entry["message"] = line
            entries.push(entry)
        }
    }

    return entries
}

let entries = parse_log("application.log")
let json_output = json_stringify(entries)
write_file("parsed_log.json", json_output)
print("Parsed " + to_string(entries.len()) + " log entries")
// Parsed 3 log entries
```

Notice that this version matches one line at a time, so the newline problem from Section 25.6.2 never arises—splitting first and matching second is often the calmer path.

No Capture Groups

POSIX ERE understands parentheses, but Lattice's three regex functions—`regex_match`, `regex_find_all`, and `regex_replace`—return only whole matches and booleans. There is no way to pull out submatch 1. When you need the pieces, match narrowly (as above, with a separate pattern per field) or fall back to `split` and the string methods from Chapter 8.

25.7 The progress Module

Long-running scripts owe their user a sign of life. Lattice ships a small native module for exactly that—a terminal progress bar that renders to standard error, rate-limits itself to one repaint every 50 ms, and disappears entirely when stderr is not a terminal, so it never pollutes a redirected log.

Listing 25.45: A hand-driven progress bar

```
import "progress" as pb

print(pb.term_width())        // 80 (or whatever your terminal is)

let bar = pb.new(3, "crunching")   // total, description -> Int handle
flux i = 0
while i < 3 {
    pb.update(bar, 1)              // advance by 1
    i += 1
}
pb.finish(bar)
print("done")
```

`pb.new` returns an integer handle, not an object. `pb.update(handle, n)` advances the bar, `pb.finish` retires it, and `pb.term_width()` reports the current terminal width so you can size your own output to match.

The Alias Form Is Required

`import "progress" as pb` works. A bare `import "progress"` does not: the stack VM and `--regvm` leave you with `undefined variable 'progress'`, and `--tree-walk` refuses outright with `import requires 'as <name>' or selective '{ ... } from'`. Always name the module.

There is also a convenience wrapper, `pb.each(array, desc, fn)`, which sizes the bar to the array, runs the callback once per element, and ticks between iterations:

Listing 25.46: Iterating with a progress bar

```
import "progress" as pb

flux total = 0
pb.each([1, 2, 3, 4], "summing", |n| {
    total = total + n
})
print("total = " + to_string(total))  // total = 10
```

`pb.each` Crashes the Tree-Walker

That listing is correct on the default stack VM and under `--regvm`. Under `--tree-walk` in v0.6.4 it terminates the process with a segmentation fault before printing anything—reliably, every run. The handle-based `new`/`update`/`finish` trio behaves on all three backends, so prefer it in code that has to run everywhere.

Exercises

1. **Stopwatch.** Write a stopwatch utility with functions `start()`, `lap()`, and `stop()` that records split times. Print each lap time and the total elapsed time, formatted as `MM:SS.mmm`.

2. **Password Hasher.** Write a function that takes a password and a salt (generated with `random_bytes`), computes `hmac_sha256(salt, password)`, and returns a string in the format `salt$hash` (where the salt is hex-encoded). Write a companion `verify` function that checks a password against a stored hash.

3. **Cron Expression Evaluator.** Write a function that takes a simple cron-like expression (e.g., `"*/5 * * * *"` for "every 5 minutes") and the current time, and returns the next time the expression will fire. Use the `time_*` extraction functions and date arithmetic.

4. **System Information Report.** Write a program that collects system information—platform, hostname, PID, current directory, all environment variables, and the current time—and outputs it as a formatted YAML document.

5. **Text Statistics.** Write a program that reads a text file and uses `regex_find_all` to count: (a) the number of words, (b) the number of sentences (ending with . ! or ?), (c) the number of email addresses, and (d) the number of URLs. Output the results as a JSON object.

6. **Injection-Proof Wrapper.** Write a function `run(program: String, argv: Array) -> Map` that calls `exec_argv` with a two-second timeout and a 1 MB stdout cap, converts a limit violation

into a Map with an `"error"` key rather than letting it escape, and passes a non-zero exit code through untouched. Then prove to yourself that `run("printf", ["%s", "; rm -rf /"])` is harmless.

7. **Progress-Reporting Hasher.** Walk a directory, compute the `sha256` of every file, and drive a `progress` bar with `new`/`update`/`finish` as you go. Confirm that redirecting stderr to a file leaves the output clean.

8. **Monte Carlo Pi Estimator.** Use `random()` to estimate π by the Monte Carlo method: generate random (x, y) points in the unit square, count how many fall inside the unit circle ($x^2 + y^2 \leq 1$), and compute $\pi \approx 4 \times \text{inside}/\text{total}$. How many points do you need for 4 decimal places of accuracy?

What's Next

With this chapter, we have completed our tour of the standard library. You now have the tools to work with files, parse and generate data in multiple formats, communicate over the network, manipulate dates, hash and encode data, generate unique identifiers, interact with the operating system, run external programs without a shell in the way, keep a user informed while you work, do mathematics, and match text with regular expressions. In Chapter 26, we will shift gears and explore Lattice's iterator system—a powerful abstraction for processing sequences lazily and composing transformations with a functional style.

Part VII

Iterators and Functional Style

Chapter 26

Lazy Iterators

Imagine you have a log file with ten million lines, and you need the first twenty that contain the word `"ERROR"`. One approach: read all ten million lines into an array, filter it, then slice off the first twenty. That works, but it loads the entire file into memory and does far more work than necessary.

There is a better way. *Iterators* let you process data one element at a time, on demand, without ever building the full collection in memory. In Lattice, iterators are *lazy*—they produce values only when asked. This chapter shows you how to create iterators, chain transformations, consume results, and build your own custom iterators from scratch.

26.1 The Iterator Protocol

At its heart, an iterator is anything that can produce a sequence of values, one at a time, until it is exhausted. Lattice's built-in `iter()` function converts a collection into an `Iterator` value, and the `.next()` method advances it by one step.

Listing 26.1: The basic iterator protocol

```
let colors = ["crimson", "teal", "gold"]
let it = iter(colors)

print(it.next())  // crimson
print(it.next())  // teal
print(it.next())  // gold
print(it.next())  // nil
```

Each call to `.next()` returns the next value. When the iterator is exhausted, it returns `nil`.

Iterator

An *iterator* is a value of type `Iterator` that encapsulates:

1. Some internal **state** (a position, a counter, a reference to a source).
2. A **next function** that, given the current state, either produces the next value or signals that the sequence is done.

Once an iterator is exhausted, every subsequent call to `.next()` returns `nil`. Iterators are single-pass: you cannot rewind them.

26.1.1 What Can You Iterate Over?

The `iter()` function accepts arrays, maps, sets, strings, ranges, and even existing iterators (which it returns unchanged):

Listing 26.2: Creating iterators from different collections

```
// Array: yields elements in order
let arr_it = iter([10, 20, 30])

// String: yields individual characters
let str_it = iter("hello")
print(str_it.next())  // h
print(str_it.next())  // e

// Range: yields integers lazily
let range_it = iter(0..5)
print(range_it.next())  // 0
print(range_it.next())  // 1

// Map: yields keys
let scores = Map::new()
scores.set("alice", 95)
scores.set("bob", 87)
let map_it = iter(scores)
print(map_it.next())  // bob
```

That last line deserves a word of warning. Map iteration walks the hash table, not the insertion list, so `"bob"` comes out first even though we stored `"alice"` first. The order is stable for a given set of keys, but it is not the order you typed them in—never write code that depends on it.

Iterators Are Single-Pass

Once you call `.next()` and consume a value, it is gone. If you need to iterate over the same data twice, create a new iterator with `iter()` again—or `.collect()` the results into an array first.

Iterators Stay on Their Thread

An iterator is stateful machinery, not plain data. As of v0.5.0, any value whose graph contains an iterator is ineligible for channel send—freezing it does not help. The attempt fails with the runtime error `cannot send a value containing an iterator on a channel`, identically on all three backends. If you need to hand results to another task, `.collect()` into an array first and send that.

26.1.2 Using Iterators in For Loops

You rarely call `.next()` by hand. Lattice's `for` loop consumes any iterable automatically:

Listing 26.3: Iterators and for loops

```
let temperatures = [72.1, 68.4, 75.0, 71.3]

// The for loop calls iter() and .next() behind the scenes
for temp in temperatures {
    print("Reading: " + to_string(temp))
}
```

When you write `for x in collection`, Lattice creates an iterator from the collection and repeatedly calls `.next()` until it returns `nil`. This is the same protocol you just learned—the `for` loop is syntactic sugar for the manual `.next()` loop.

26.2 Sources: Where Iterators Come From

The `iter()` function wraps existing collections, but Lattice also provides dedicated *source* constructors that create iterators from scratch.

26.2.1 iter() — From Collections

As we saw above, `iter()` is the universal entry point. It accepts arrays, maps, sets, strings, ranges, and iterators. Under the hood, `iter()` dispatches to specialized constructors like `iter_from_array`, `iter_from_map`, and `iter_from_range` in the C runtime (see `src/iterator.c`).

26.2.2 range_iter() — Lazy Integer Sequences

While `iter(0..1000)` converts a range to an iterator, the dedicated `range_iter()` function gives you more control—including a custom step:

Listing 26.4: Creating range iterators

```
// Count from 0 to 9
let digits = range_iter(0, 10)
print(digits.collect())  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

// Even numbers from 0 to 20
let evens = range_iter(0, 20, 2)
print(evens.collect())  // [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

// Countdown
let countdown = range_iter(5, 0, -1)
print(countdown.collect())  // [5, 4, 3, 2, 1]
```

The third argument is the step. If omitted, it defaults to `1` for ascending ranges or `-1` for descending ranges. No array is allocated—the iterator computes each value on the fly.

When to Use range_iter() vs. a Range Literal

If you need a custom step or you are building a pipeline of iterator transformations, use `range_iter()`. For straightforward `for` loops, the range literal (`0..10`) is more concise and equally lazy.

26.2.3 repeat_iter() — The Same Value, Over and Over

Need a sequence of identical values? `repeat_iter()` yields the same value a specified number of times—or infinitely if you omit the count:

Listing 26.5: Repeating a value

```
// Repeat a value 4 times
let zeros = repeat_iter(0, 4)
print(zeros.collect())  // [0, 0, 0, 0]

// Infinite repetition (be careful!)
let ones = repeat_iter(1)
// ones.collect() would run forever --- use .take() instead
print(ones.take(5).collect())  // [1, 1, 1, 1, 1]
```

Infinite Iterators

Calling `.collect()` on an infinite iterator will consume all available memory and hang your program. Always pair infinite sources with a limiting transformer like `.take()` before consuming.

26.3 Transformers: Reshaping the Stream

The real power of iterators emerges when you chain *transformers*—methods that wrap one iterator in another, building up a processing pipeline without doing any work until you ask for results.

26.3.1 .map() — Transform Each Element

The `.map()` method applies a closure to every element the iterator produces:

Listing 26.6: Mapping over an iterator

```
let prices = [19.99, 45.50, 12.00, 89.95]
let with_tax = iter(prices)
    .map(|price| { price * 1.08 })
    .collect()

print(with_tax)
// [21.5892, 49.14, 12.96, 97.146]
```

The closure receives each value and returns its transformation. Because `.map()` returns a new iterator, no work happens until `.collect()` pulls values through.

Tapping a Pipeline

A map callback may also capture an accumulator from the enclosing scope, which gives you a cheap way to watch values go by:

```
flux seen = []
let tap = |x| { seen.push(x); x * 2 }

print(iter([1, 2, 3]).map(tap).collect())  // [2, 4, 6]
print(seen)                                // [1, 2, 3]
```

Mutating a captured container from inside a closure became reliable in v0.6.4—`.push()` on an upvalue now writes back on every backend. Older code worked around this with pre-allocated arrays and manual index bookkeeping; that ceremony is no longer necessary.
Keep the effects observational, though. On the bytecode backends nothing is recorded until `.collect()` pulls, while `--tree-walk` runs the callbacks as the chain is built—so a pipeline whose side effects you *depend* on will not behave the same everywhere. Tap to watch; do not tap to compute.

26.3.2 .filter() — Keep Only What Matters

The `.filter()` method takes a predicate closure and yields only the elements for which it returns `true`:

Listing 26.7: Filtering an iterator

```
let readings = [3.1, 7.8, 2.4, 9.0, 5.5, 1.2]
let high_readings = iter(readings)
    .filter(|r| { r > 5.0 })
    .collect()

print(high_readings)  // [7.8, 9, 5.5]
```

(Lattice prints whole-number floats without a decimal point, which is why `9.0` appears as `9`.) Internally, the filter iterator loops through the inner source, skipping elements that don't match, until it finds one that does—or the source is exhausted. You can see this logic in `src/iterator.c` where `iter_filter_next` calls the inner iterator in a loop.

26.3.3 .take() — Limit the Count

The `.take(n)` transformer yields at most `n` elements, then stops:

Listing 26.8: Taking the first few elements

```
let first_three = range_iter(0, 1000000)
    .take(3)
    .collect()

print(first_three)  // [0, 1, 2]
```

Despite the range spanning a million integers, only three are ever produced. This is laziness in action: the remaining 999,997 values are never computed.

26.3.4 .skip() — Jump Ahead

The companion to `.take()`, the `.skip(n)` transformer discards the first `n` elements and yields the rest:

Listing 26.9: Skipping elements

```
let after_header = iter(["name", "age", "city", "Alice", "30", "Portland"])
    .skip(3)
    .collect()

print(after_header)  // [Alice, 30, Portland]
```

You can combine `.skip()` and `.take()` to extract a window from a sequence:

Listing 26.10: Pagination with skip and take

```
let all_items = range_iter(1, 101)   // items 1 through 100
let page_size = 10
let page_number = 3

let page = all_items
    .skip((page_number - 1) * page_size)
    .take(page_size)
    .collect()

print(page)  // [21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
```

26.3.5 .enumerate() — Track the Index

When you need both the position and the value, `.enumerate()` wraps each element in a two-element array `[index, value]`:

Listing 26.11: Enumerating an iterator

```
let languages = ["Lattice", "Rust", "Python", "Go"]
let indexed = iter(languages).enumerate()

for pair in indexed.collect() {
    print(to_string(pair[0]) + ": " + pair[1])
}
// 0: Lattice
// 1: Rust
// 2: Python
// 3: Go
```

The index always starts at zero.

26.3.6 .zip() — Walk Two Iterators in Lockstep

The `.zip()` transformer pairs elements from two iterators into `[left, right]` arrays. It stops as soon as either iterator is exhausted:

Listing 26.12: Zipping two iterators

```
let names = iter(["Alice", "Bob", "Carol"])
let scores = iter([95, 87, 92])

let paired = names.zip(scores).collect()
print(paired)
// [[Alice, 95], [Bob, 87], [Carol, 92]]
```

Zip is especially useful for combining parallel data sources:

Listing 26.13: Merging labels with values

```
let labels = iter(["Temperature", "Humidity", "Pressure"])
let values = iter([72.1, 45.0, 1013.25])
let units = iter(["F", "%", "hPa"])

// Zip labels with values, then map to formatted strings
let report = labels.zip(values).zip(units)
    .map(|entry| {
        let label_value = entry[0]
        let unit = entry[1]
        label_value[0] + ": " + to_string(label_value[1]) + " " + unit
    })
    .collect()

for line in report {
    print(line)
}
// Temperature: 72.1 F
// Humidity: 45 %
// Pressure: 1013.25 hPa
```

26.3.7 Chaining Transformers

Transformers return iterators, so you can chain them into expressive pipelines:

Listing 26.14: A multi-step pipeline

```
let words = ["Lattice", "is", "a", "language", "for", "the", "curious"]

let result = iter(words)
    .filter(|w| { len(w) > 2 })
    .map(|w| { w + "!" })
    .enumerate()
    .take(3)
    .collect()

print(result)
// [[0, Lattice!], [1, language!], [2, for!]]
```

No intermediate arrays are created. Each value flows through the entire chain—from source to `.filter()` to `.map()` to `.enumerate()` to `.take()`—one at a time, on demand.

Reading Pipelines

Read iterator chains from top to bottom, like a recipe:

1. Start with the words.
2. Keep only words longer than two characters.
3. Append an exclamation mark.
4. Number them.
5. Take the first three.
6. Collect into an array.

26.4 Consumers: Triggering the Work

Transformers are lazy—they build up a chain of work but produce no output on their own. *Consumers* are the methods that actually pull values through the pipeline and produce a final result.

26.4.1 .collect() — Materialize into an Array

We have been using `.collect()` throughout this chapter. It drains the iterator and gathers every element into an array:

Listing 26.15: Collecting an iterator

```
let squares = range_iter(1, 6)
    .map(|n| { n * n })
    .collect()

print(squares)  // [1, 4, 9, 16, 25]
```

Under the hood, `iter_collect` in `src/iterator.c` starts with a small buffer (capacity 8) and doubles it as needed, appending each value from `iter_next` until the iterator signals done. The alias `.to_array()` does exactly the same thing.

26.4.2 .reduce() — Fold into a Single Value

The `.reduce()` method collapses an iterator into a single accumulated value. It takes a closure and an initial accumulator:

Listing 26.16: Reducing an iterator

```
let total = iter([10, 20, 30, 40])
    .reduce(|acc, val| { acc + val }, 0)

print(total)  // 100
```

The closure receives two arguments: the running accumulator and the current element. The second argument to `.reduce()` is the initial value for the accumulator.

Here is a more involved example—computing the maximum value:

Listing 26.17: Finding the maximum with reduce

```
let temperatures = [68.2, 71.5, 65.0, 73.8, 70.1]

let hottest = iter(temperatures)
    .reduce(|max_temp, temp| {
        if temp > max_temp { temp } else { max_temp }
    }, temperatures[0])

print(hottest)  // 73.8
```

26.4.3 .any() and .all() — Short-Circuit Boolean Tests

These two consumers answer yes-or-no questions about the sequence. Both *short-circuit*: they stop consuming as soon as the answer is determined.

Listing 26.18: Testing with .any() and .all()

```
let grades = [88, 92, 75, 95, 61]

let has_failing = iter(grades).any(|g| { g < 65 })
print(has_failing)  // true (stops after finding 61)

let all_passing = iter(grades).all(|g| { g >= 60 })
print(all_passing)  // true
```

The `.any()` consumer returns `true` as soon as it finds a matching element, without examining the rest. The `.all()` consumer returns `false` as soon as it finds a non-matching element. If the iterator is empty, `.any()` returns `false` and `.all()` returns `true`—consistent with the mathematical conventions for existential and universal quantification.

26.4.4 .count() — How Many?

The `.count()` consumer drains the iterator and returns the number of elements:

Listing 26.19: Counting elements

```
let long_words = iter(["cat", "elephant", "dog", "rhinoceros", "ox"])
    .filter(|w| { len(w) > 4 })
    .count()

print(long_words)  // 2
```

Consumers Are Destructive

Calling a consumer like `.collect()`, `.reduce()`, or `.count()` drains the iterator. If you try to use the same iterator afterwards, it will produce no more values. To reuse, build a fresh iterator or collect into an array first.

26.4.5 Summary of Consumers

Table 26.1: Iterator consumer methods

Method	Returns	Description
`.collect()`	`Array`	Gather all elements into an array
`.to_array()`	`Array`	Alias for `.collect()`
`.reduce(fn, init)`	`Any`	Fold with accumulator
`.any(predicate)`	`Bool`	True if any element matches
`.all(predicate)`	`Bool`	True if all elements match
`.count()`	`Int`	Count elements (consumes all)

26.5 When a Callback Fails

The closures you hand to `.map()` and friends are ordinary Lattice code, and ordinary Lattice code can fail. Since v0.6.4, what happens next is well defined: the failure propagates out of the callback, the pipeline stops where it stood, and the error surfaces at the consumer that was pulling values.

Listing 26.20: A callback that divides by zero

```
let batch = [12, 4, 0, 3]

let inverted = iter(batch)
    .map(|n| { 60 / n })
    .collect()

print(inverted)
```

Nothing prints. Instead the program stops with the callback's own error:

```
vm error: division by zero
stack trace (most recent call last):
  [line 2] in <script>
  [line 0] in <closure>
  [line 3] in <closure>
```

The two `<closure>` frames are the map callback and the pipeline that called it. On `--tree-walk` the same message appears with the `error:` prefix, and on `--regvm` with `regvm error:`—the core text is identical everywhere.

The dependable cure is to keep bad values out of the pipeline in the first place. A `.filter()` in front of the `.map()` costs nothing, because both stages are lazy:

Listing 26.21: Guarding a pipeline instead of catching

```
let batch = [12, 4, 0, 3]

let inverted = iter(batch)
    .filter(|n| { n != 0 })
    .map(|n| { 60 / n })
    .collect()

print(inverted)  // [5, 15, 20]
```

Do Not Wrap a Lazy Pipeline in try/catch

Catching a callback failure works reliably for the *eager* array method `Array.map()`:

```
try {
    let inverted = batch.map(|n| { 60 / n })
    print(inverted)
} catch e {
    print("bad batch: " + e.message)   // bad batch: division by zero
}
```

That form is verified on all three backends. Wrapping a *lazy* `iter(...)` chain—or `Array.filter()` or `Array.reduce()`—in `try`/`catch` is a different story: as of v0.6.4 the unwind does not reach the handler on the bytecode backends, and the interpreter stops the process instead. Only `--tree-walk` recovers. Until that is harmonized, guard the data before the pipeline rather than catching after it.

26.6 Building Your Own Iterators

The built-in iterator methods cover the common cases, but sometimes you need a custom sequence—Fibonacci numbers, lines from a database cursor, or values generated by a complex algorithm.

Lattice's `lib/fn.lat` standard library module provides a protocol for building custom lazy sequences entirely in Lattice code. Import it, and you get access to a rich set of sequence constructors and transformers.

Where Does `lib/fn` Come From?

The path `"lib/fn"` is a relative import, and on the default backend relative imports resolve against the *current working directory* (see Section 28.5). The examples in this chapter assume you run `clat` from the root of the Lattice source tree, where `lib/fn.lat` lives. For your own projects, copy `lib/fn.lat` into the project tree and adjust the import path accordingly.

26.6.1 The Sequence Protocol

A lazy sequence in the `fn` module is a map with two keys:

- `"state"`: the current position or state of the iterator (any value).
- `"next"`: a function that takes the current state and returns a step map.

Each *step* is itself a map:

- `\{"value": v, "done": false, "state": next_state\}` — yields value `v` and advances.
- `\{"done": true\}` — the sequence is exhausted.

Lazy Sequence (fn module)

A lazy sequence threads state explicitly through each step. This design is compatible with Lattice's value semantics: no hidden mutable references, no shared state. The next function is pure—given the same state, it always produces the same step.

26.6.2 Building a Fibonacci Iterator

Let's build a lazy Fibonacci sequence using the `fn` module:

Listing 26.22: A custom Fibonacci iterator

```
import "lib/fn" as F

// State is [current, next_val]
let fibs = F.iterate([0, 1], |pair| {
    [pair[1], pair[0] + pair[1]]
})

// iterate() yields the state at each step,
// so we fmap to extract just the first element
let fib_numbers = F.fmap(fibs, |pair| { pair[0] })

// Take the first 10 Fibonacci numbers
let result = F.collect(F.take(fib_numbers, 10))
print(result)  // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

The key function here is `F.iterate(seed, f)`, which produces an infinite sequence: `seed`, `f(seed)`, `f(f(seed))`, and so on. We start with `[0, 1]` and advance by shifting the pair forward.

26.6.3 Custom Sequences from Scratch

You can also build a sequence manually by providing a state and a next function. Here is a sequence that yields powers of two:

Listing 26.23: Building a sequence from scratch

```
import "lib/fn" as F

fn powers_of_two() -> Map {
    // State is the current power
    let initial_state = 1
    let next_fn = |current| {
        let step = Map::new()
        step.set("value", current)
        step.set("done", false)
        step.set("state", current * 2)
        return step
    }

    let seq = Map::new()
    seq.set("state", initial_state)
    seq.set("next", next_fn)
    return seq
}

let result = F.collect(F.take(powers_of_two(), 8))
print(result)  // [1, 2, 4, 8, 16, 32, 64, 128]
```

The pattern is always the same: define an initial state, write a function that takes the current state and returns a step map, then combine them into a sequence map.

26.6.4 A Practical Example: Paginated API Results

Custom iterators shine when the data source is external or unbounded. Here is a sketch of a lazy sequence that fetches pages from an API:

Listing 26.24: Lazy pagination (conceptual)

```
import "lib/fn" as F

fn paginated_users(base_url: String) -> Map {
    // State is the current page number
    return F.iterate(1, |page| { page + 1 })
}

// In a real app, you would use fmap to fetch each page:
// let pages = F.fmap(paginated_users("/api/users"), |page| {
//     http_get(base_url + "?page=" + to_string(page))
// })
//
// Then take pages until the response is empty.
```

The lazy approach means you never fetch a page you don't need.

26.6.5 Composing fn Module Sequences

The `fn` module provides the same family of transformers as the built-in iterator methods, but as standalone functions that work with the sequence protocol:

Listing 26.25: Composing sequences with the fn module

```
import "lib/fn" as F

// Start with a range, filter, transform, and collect
let result = F.collect(
    F.take(
        F.fmap(
            F.select(F.range(1, 100), |x| { x % 3 == 0 }),
            |x| { x * x }
        ),
        5
    )
)

print(result)  // [9, 36, 81, 144, 225]
```

This computes the squares of the first five multiples of three. Every step is lazy—the range never produces 100 values, only as many as needed to fill the first five matches.

Table 26.2: fn module sequence functions

Function	Kind	Description
`F.range(start, end, step?)`	Source	Lazy integer range
`F.from_array(arr)`	Source	Lazy sequence from array
`F.repeat(value)`	Source	Infinite repetition
`F.iterate(seed, f)`	Source	Infinite `seed, f(seed), ...`
`F.fmap(seq, f)`	Transformer	Transform each element
`F.select(seq, pred)`	Transformer	Keep matching elements
`F.take(seq, n)`	Transformer	First `n` elements
`F.drop(seq, n)`	Transformer	Skip first `n`
`F.take_while(seq, pred)`	Transformer	Yield while predicate holds
`F.zip(seq1, seq2)`	Transformer	Pair elements from two sequences
`F.collect(seq)`	Consumer	Collect into array
`F.fold(seq, init, f)`	Consumer	Reduce with accumulator
`F.count(seq)`	Consumer	Count elements

Two Iterator Systems

Lattice has two complementary iteration systems:

1. **Built-in iterators** (`iter()`, `.map()`, `.filter()`, etc.) — implemented in C (`src/iterator.c`), accessed via method chaining. Best for everyday use.
2. **fn module sequences** (`F.range()`, `F.fmap()`, `F.select()`, etc.) — implemented in pure Lattice (`lib/fn.lat`), using a map-based protocol. Best for custom sequences and functional composition.

Both are lazy. The built-in system is faster (native C); the `fn` module is more flexible (user-extensible).

26.7 Why Laziness Matters

We have seen iterators in action, but let's make the case for *why* you should prefer them over eager array operations.

26.7.1 Memory Efficiency

Consider processing a large dataset:

Listing 26.26: Eager vs. lazy memory usage

```
// Eager: creates three intermediate arrays
let data = range_iter(0, 1000000).collect()  // 1M element array
let filtered = data.filter(|x| { x % 7 == 0 })  // another array
let mapped = filtered.map(|x| { x * x })  // yet another
let result = mapped[0..5]  // finally, what we wanted

// Lazy: creates zero intermediate arrays
let result_lazy = range_iter(0, 1000000)
    .filter(|x| { x % 7 == 0 })
    .map(|x| { x * x })
    .take(5)
    .collect()

print(result_lazy)  // [0, 49, 196, 441, 784]
```

The eager version allocates three full arrays before extracting five values. The lazy version allocates one small array (the final result) and pulls only the 29 integers—0 through 28—needed to find five multiples of seven.

26.7.2 Computation Efficiency

Laziness also saves computation. When `.take(5)` has collected its five elements, the entire pipeline stops—no more filtering, no more mapping, and the range never produces its remaining elements.

We can measure that with a counter:

Listing 26.27: Counting how much work laziness avoids

```
flux calls = 0
let bump = |x| { calls = calls + 1; x * x }

let out = range_iter(0, 1000000).map(bump).take(3).collect()

print(out)    // [0, 1, 4]
print(calls)  // 3
```

Three calls out of a million candidates. This is especially valuable when the source is expensive (network calls, file I/O) or the filter is highly selective.

Laziness Is a Bytecode-Backend Guarantee

That counter reads `3` on the default stack VM and on `--regvm`. Run the same program under `--tree-walk` and it reads `1000000`: the tree-walk evaluator applies `.map()` and `.filter()` eagerly across the whole source before `.take()` trims the result.
The answers agree—`[0, 1, 4]` either way—but the work does not. When you are reasoning about cost, or when a callback touches the network, run on the default backend.

26.7.3 Infinite Sequences

Some sequences have no natural end—the natural numbers, a stream of sensor readings, lines from a log file that is still being written. Lazy iterators handle these gracefully because they never try to materialize the whole sequence:

Listing 26.28: Working with infinite sequences

```
import "lib/fn" as F

// The natural numbers
let naturals = F.range(1, 9999999)

// First 5 perfect squares
let squares = F.collect(
    F.take(
        F.fmap(naturals, |n| { n * n }),
        5
    )
)
print(squares)  // [1, 4, 9, 16, 25]

// First 5 numbers divisible by both 3 and 7
let div_3_and_7 = F.collect(
    F.take(
        F.select(F.range(1, 9999999), |n| { n % 3 == 0 && n % 7 == 0 }),
        5
    )
)
print(div_3_and_7)  // [21, 42, 63, 84, 105]
```

26.7.4 When to Stay Eager

Laziness is not always the right choice. If you need to access elements by index, sort the collection, or iterate over the data multiple times, collecting into an array first is the pragmatic approach. Iterators are single-pass and sequential—they trade random access for efficiency.

A Rule of Thumb

Use iterators when you are *flowing* data through a pipeline: read, transform, consume. Use arrays when you need to *hold* data: index, sort, search, iterate multiple times.

26.8 Under the Hood

If you are curious about how iterators work at the C level, here is a brief tour.

Each built-in iterator is a `LatValue` of type `VAL_ITERATOR`. It stores three things:

- A **state pointer** (`void *state`)—an opaque blob of data specific to the iterator kind (array index, range counter, wrapped inner iterator, etc.).
- A **next function** (`next_fn`)—a C function pointer that advances the state and returns the next value.
- A **free function** (`free_fn`)—cleans up the state when the iterator is no longer needed.

The `iter_next` call (defined in `include/iterator.h`) is a single function pointer dispatch:

```
return iter->as.iterator.next_fn(iter->as.iterator.state, done);
```

Transformer iterators like `iter_take` wrap an inner iterator by storing it in their state struct (e.g., `IterTakeState` contains the inner iterator and a remaining count). When their `next_fn` is called, they delegate to the inner iterator's `next_fn`, applying their transformation. This forms a chain of function pointer calls—lightweight and allocation-free during iteration.

Iterator Ownership

Transformer constructors like `iter_take()` and `iter_map_transform()` *take ownership* of the inner iterator. Once you pass an iterator to a transformer, you should not use the original. The built-in method-chaining API handles this naturally—each `.take()` or `.map()` call consumes the receiver and returns a new iterator.

26.9 Exercises

1. **Sum of Squares.** Use `range_iter`, `.map()`, and `.reduce()` to compute the sum of squares of the integers from 1 to 100. What answer do you get?

2. **Custom Collatz Iterator.** The Collatz sequence for a starting number `n` is: if `n` is even, the next value is `n / 2`; if odd, it is `3 * n + 1`. The sequence ends when it reaches 1. Using the `fn` module's `iterate()` and `take_while()`, build a lazy Collatz sequence starting from 27 and collect it into an array. How many steps does it take?

3. **Zip and Reduce.** Given two arrays of equal length—one of item names and one of prices—use `iter()`, `.zip()`, and `.reduce()` to compute the total price. For example: names `["apple", "bread", "milk"]` and prices `[1.20, 3.50, 2.80]` should yield `7.50`.

4. **Infinite Primes.** Write a function that returns a lazy sequence (using the `fn` module) of prime numbers. Use it to print the first 20 primes. *Hint*: use `F.select` with a primality-testing closure over `F.range(2, 9999999)`.

5. **Interleave.** Write a function `interleave(seq1, seq2)` that takes two lazy sequences and produces a new sequence that alternates elements: first from `seq1`, then from `seq2`, then `seq1`, and so on. Stop when either sequence is exhausted.

What's Next Iterators give you a powerful way to express data transformations as pipelines. In the next chapter, we take this further: Chapter 27 introduces Lattice's functional programming tools—`pipe()`, `compose()`, currying, and the full `fn` standard library—that let you build elegant, composable abstractions on top of the iterator foundation we have laid here.

Chapter 27

Functional Programming in Lattice

Lattice is not a purely functional language—it has mutable variables, side effects, and imperative loops. But it provides a rich set of functional tools that, when used well, can make your code more concise, more composable, and easier to reason about.

In Chapter 26, we built pipelines of iterator transformations. This chapter takes those ideas further. We will explore Lattice's built-in functional primitives—`pipe()`, `compose()`, and `identity()`—then dive into the `fn` standard library's currying, partial application, and collection utilities. By the end, you will know when functional style shines in Lattice and when to reach for something else.

27.1 pipe(), compose(), and identity()

These three built-in functions form the foundation of functional composition in Lattice. They are implemented directly in the runtime (`src/runtime.c` and `src/eval.c`), so they are always available without any imports.

27.1.1 pipe() — Threading Values Through Functions

The `pipe()` function takes a value and one or more functions, then threads the value through each function left to right:

Listing 27.1: Basic pipe usage

```
let result = pipe(5,
    |x| { x * 2 },
    |x| { x + 1 },
    |x| { x * x }
)
print(result)  // 121  (5 -> 10 -> 11 -> 121)
```

Each function receives the output of the previous one. The first argument is the initial value; every subsequent argument must be a closure.

pipe()

`pipe(val, f1, f2, ..., fn)` computes `fn(...f2(f1(val)))`. It evaluates left to right: the value flows *forward* through the chain, like water through a series of pipes.

Without `pipe()`, you would write deeply nested function calls:

Listing 27.2: Nested calls vs. pipe

```
// Without pipe: read inside-out
let result = square(increment(double(5)))

// With pipe: read top-to-bottom
let result = pipe(5,
    |x| { x * 2 },
    |x| { x + 1 },
    |x| { x * x }
)
```

The pipe version reads in the order that operations happen—no mental stack unwinding required.

Here is a more realistic example, processing a string:

Listing 27.3: String processing with pipe

```
let raw_input = "  Hello, Lattice World!  "

let processed = pipe(raw_input,
    |s| { s.trim() },
    |s| { s.to_lower() },
    |s| { s.replace(",", "") },
    |s| { s.split(" ") }
)

print(processed)  // [hello, lattice, world!]
```

pipe() with Named Functions

You are not limited to anonymous closures. Named functions work too:

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

let result = pipe(5, double, increment)
print(result)  // 11
```

One caveat, and it bites in more places than `pipe()`: a `fn` declaration is only a first-class value on the bytecode backends. Under `--tree-walk`, writing `double` where a value is expected reports `undefined variable 'double'`, because the tree-walk evaluator resolves `fn` names at call sites only. A closure bound with `let`—`let double = |x| { x * 2 }`—is a value on all three backends, so that is the portable form.

27.1.2 compose() — Building New Functions from Old

While `pipe()` immediately applies a chain of functions to a value, `compose()` builds a *new function* from two existing ones—without calling either:

Listing 27.4: Composing functions

```
let double = |x| { x * 2 }
let increment = |x| { x + 1 }

// compose(f, g) returns a new function: |x| { f(g(x)) }
let double_then_increment = compose(increment, double)

print(double_then_increment(5))   // 11  (double(5)=10, increment(10)=11)
print(double_then_increment(10))  // 21  (double(10)=20, increment(20)=21)
```

Argument Order

`compose(f, g)` applies `g` first, then `f`. In mathematical notation: `compose(f, g)(x) = f(g(x))`. This is right-to-left, like function composition in mathematics. If you find this confusing, use `pipe()` instead—it flows left-to-right.

The value of `compose()` is that it creates a reusable function you can pass around, store in variables, or use as a callback:

Listing 27.5: Composed functions as callbacks

```
let to_celsius = |f| { (f - 32) * 5 / 9 }
let round_temp = |c| { to_int(c) }
let format_temp = |c| { to_string(c) + " C" }

// Build a single conversion function
let convert = compose(format_temp, compose(round_temp, to_celsius))

let temps_f = [98.6, 212.0, 32.0, 72.5]
let temps_c = iter(temps_f).map(convert).collect()
print(temps_c)  // [37 C, 100 C, 0 C, 22 C]
```

Under the hood, `compose()` stashes `f` and `g` under generated names and builds a one-parameter closure around them. In the tree-walk evaluator (`src/eval.c`) that closure is an AST node for `__compose_f(__compose_g(x))` over a cloned environment. In the stack VM (`src/runtime.c`, `native_compose`) it is a six-instruction chunk: fetch `f`, fetch `g`, load the argument, call `g`, call `f` on the result, return. Either way, the composed function behaves like any other closure.

A Silent `nil` Means "Not a Closure"

Both `pipe()` and `compose()` check that the functions handed to them really are closures. When the check fails on the bytecode backends they return `nil` rather than raising:

```
print(pipe(5, 3))       // nil
print(compose(3, 4))    // nil
```

`--tree-walk` is stricter and reports `pipe() argument 2 is not a function` or `compose() arguments must be closures`. If a composition chain unexpectedly evaluates to `nil` on the default backend, the first thing to check is whether every stage is really a closure.

27.1.3 identity() — The Do-Nothing Function

The `identity()` function returns its argument unchanged:

Listing 27.6: The identity function

```
print(identity(42))        // 42
print(identity("hello"))   // hello
print(identity([1, 2, 3])) // [1, 2, 3]
```

At first glance, this seems useless. Why would you want a function that does nothing? The answer is that `identity()` serves as a *default* or *neutral element* in functional pipelines:

Listing 27.7: identity() as a default transform

```
fn process_data(values: Array, transform: Fn) -> Array {
    return iter(values).map(transform).collect()
}

// Sometimes we want a transform, sometimes we don't
let raw = [1, 2, 3, 4, 5]
let unchanged_fn = |x| { identity(x) }

let doubled = process_data(raw, |x| { x * 2 })
print(doubled)  // [2, 4, 6, 8, 10]

let unchanged = process_data(raw, unchanged_fn)
print(unchanged)  // [1, 2, 3, 4, 5]
```

Rather than adding special-case logic for "no transformation," you pass a neutral transform. The code stays uniform.

The same trick makes `identity` the neutral element for `compose()`. Composing any function with it gives back the original function:

Listing 27.8: identity() as composition neutral

```
let double = |x| { x * 2 }
let keep = |x| { identity(x) }

let same = compose(double, keep)
print(same(5))  // 10
```

Wrap `identity` Before You Pass It

Notice that both examples wrap the builtin in a closure—`|x| { identity(x) }`—rather than handing over the bare name. That is deliberate. `identity` is a runtime builtin, not an ordinary binding, and as of v0.6.4 passing it *by name* into a higher-order position does not work uniformly:

- Under `--tree-walk`, `identity` in value position reports `undefined variable 'identity'`.
- On the default stack VM, `iter(values).map(identity).collect()` yields `[nil, nil, nil]` and `pipe(5, identity)` yields `nil`.
- Only `--regvm` passes the bare builtin through intact.

Calling `identity(x)` directly is fine everywhere. The one-line wrapper turns the builtin into a real closure, and real closures travel without incident on all three backends.

27.2 The fn Standard Library Module

The `fn` module (`lib/fn.lat`) bundles a collection of functional programming utilities into a single import. We encountered its lazy sequence functions in Chapter 26; now let's explore the rest.

Listing 27.9: Importing the fn module

```
import "lib/fn" as F
```

27.2.1 Function Composition Utilities

The module provides several utilities for building and combining functions:

Listing 27.10: comp, flip, and constant

```
import "lib/fn" as F

// comp(f, g) is a module-accessible wrapper around compose()
let add_one_then_double = F.comp(|x| { x * 2 }, |x| { x + 1 })
print(add_one_then_double(4))  // 10

// flip(f) reverses the argument order of a 2-argument function
let div = |a, b| { a / b }
let inv_div = F.flip(div)
print(inv_div(2, 10))  // 5  (calls div(10, 2))

// constant(v) returns a function that always returns v
let always_zero = F.constant(0)
print(always_zero(99))     // 0
print(always_zero("hi"))   // 0
```

27.2.2 apply_n — Repeated Application

The `F.apply_n(f, n, value)` function applies `f` to `value` a total of `n` times:

Listing 27.11: Repeated function application

```
import "lib/fn" as F

let double = |x| { x * 2 }
print(F.apply_n(double, 0, 3))  // 3
print(F.apply_n(double, 1, 3))  // 6
print(F.apply_n(double, 4, 1))  // 16  (1 -> 2 -> 4 -> 8 -> 16)
```

This is a controlled form of recursion—useful for simulations, numerical methods, or any situation where you want to iterate a transformation a fixed number of times.

27.2.3 thread() — Module-Level Piping

The `F.thread(value, f1, f2)` function is the module's equivalent of a two-function `pipe()`, for cases where you want to use the `fn` module's qualified syntax:

Listing 27.12: thread for 2-step piping

```
import "lib/fn" as F

let result = F.thread(10,
    |x| { x + 5 },
    |x| { x * 2 }
)
print(result)  // 30
```

For longer chains, the built-in `pipe()` is more flexible since it accepts any number of functions.

27.2.4 Collection Utilities

Beyond sequences and composition, the `fn` module includes utilities for working with arrays and maps.

group_by — Categorize Elements

Listing 27.13: Grouping elements

```
import "lib/fn" as F

let words = ["apple", "avocado", "banana", "blueberry", "cherry"]
let grouped = F.group_by(words, |w| { w[0] })

print(grouped)
// {"c": [cherry], "b": [banana, blueberry], "a": [apple, avocado]}
```

The key function determines which group each element belongs to. The result is a map from keys to arrays of matching elements. Maps print their keys in hash order rather than insertion order, which is why `"c"` leads—the grouping is correct regardless.

partition — Split into Two Groups

Listing 27.14: Partitioning an array

```
import "lib/fn" as F

let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let result = F.partition(numbers, |n| { n % 2 == 0 })

print(result[0])  // [2, 4, 6, 8, 10]  (matches)
print(result[1])  // [1, 3, 5, 7, 9]   (non-matches)
```

frequencies — Count Occurrences

Listing 27.15: Counting element frequencies

```
import "lib/fn" as F

let votes = ["yes", "no", "yes", "yes", "no", "abstain"]
let tally = F.frequencies(votes)

print(tally)
// {"yes": 3, "no": 2, "abstain": 1}
```

uniq_by — Deduplicate by Key

Listing 27.16: Deduplication by a key function

```
import "lib/fn" as F

let items = [1, 2, 3, 4, 5, 6, 7, 8, 9]
let unique_mod3 = F.uniq_by(items, |x| { x % 3 })

print(unique_mod3)  // [1, 2, 3]  (first occurrence of each mod-3 class)
```

chunk and flatten — Reshaping Arrays

Two more shape-changers round out the set. `F.chunk` slices an array into fixed-size batches (the last one may be short), and `F.flatten` collapses one level of nesting:

Listing 27.17: Batching and flattening

```
import "lib/fn" as F

print(F.chunk([1, 2, 3, 4, 5], 2))       // [[1, 2], [3, 4], [5]]
print(F.flatten([[1, 2], [3], [4, 5]]))  // [1, 2, 3, 4, 5]
```

`F.chunk` pairs well with batched I/O: chunk a work list, then process one batch per request.

27.2.5 The Result Type

The `fn` module also provides a lightweight `Result` type for representing success or failure without exceptions:

Listing 27.18: The Result type from the fn module

```
import "lib/fn" as F

let success = F.ok(42)
let failure = F.err("something went wrong")

print(F.is_ok(success))   // true
print(F.is_err(failure))  // true

print(F.unwrap(success))  // 42
print(F.unwrap_or(failure, 0))  // 0

// Transform the Ok value
let doubled = F.map_result(success, |x| { x * 2 })
print(F.unwrap(doubled))  // 84

// Chain operations that might fail
let chained = F.flat_map_result(success, |x| {
    if x > 0 { F.ok(x * 10) } else { F.err("must be positive") }
})
print(F.unwrap(chained))  // 420
```

The `F.try_fn()` wrapper catches runtime errors and returns a Result:

Listing 27.19: Catching errors with try_fn

```
import "lib/fn" as F

let safe = F.try_fn(|_| { 100 / 4 })
print(F.unwrap(safe))  // 25

let risky = F.try_fn(|_| { 100 / 0 })
print(F.is_err(risky))  // true
```

27.3 Closures That Accumulate

Pure pipelines are lovely, but plenty of real work needs a closure that remembers something between calls. A tally, a log, a growing result list—an *accumulator* captured from the enclosing scope.

Listing 27.20: An accumulator captured by a closure

```
flux collected = []
let add = |x| { collected.push(x) }

add("alpha")
add("beta")

print(collected)        // [alpha, beta]
print(collected.len())  // 2
```

That reads exactly the way you would hope, and as of v0.6.4 it behaves exactly the way it reads on all three backends.

It is worth knowing that this is recent. Before v0.6.4, a captured variable—an *upvalue*—could be reassigned (`count = count + 1`) or written through an index (`slots[i] = v`), but calling a mutating builtin method on it was a trap: the bytecode backends cloned the captured container, mutated the clone, and dropped it. `collected` would still print `[]`. Whole idioms grew up around the gap—pre-allocated arrays with a manual index counter, parallel arrays instead of maps. You will still find that shape in older Lattice code. It is no longer necessary.

What Changed in v0.6.4

Mutating builtin methods on a captured receiver now write their result back into the upvalue cell. The stack VM does this with a dedicated `OP_INVOKE_MUT` instruction (`ROP_INVOKE_MUT` on the register VM), which knows that the receiver of `.push()`, `.set()`, `.add()` and friends must be updated in place rather than consumed as a temporary. The same change is what made nested mutation—`records[0].push(x)`, `index["k"].push(x)`—start working.

Frozen receivers still refuse the write, and now they refuse it loudly. Freeze `collected` before handing it to the closure and the first `add()` stops with `cannot call mutating method 'push' on crystal value 'collected' (use thaw(collected) to make it mutable)`—the same core message on all three backends.

Because the cell itself is shared, several closures can be handed out over the same accumulator and each will see the others' work:

Listing 27.21: Two closures sharing one cell

```
fn make_logger() -> Map {
    flux entries = []
    let record = |msg| { entries.push(msg) }
    let dump = |_| { entries }

    flux api = Map::new()
    api["record"] = record
    api["dump"] = dump
    return api
}

let log = make_logger()

let record = log["record"]
record("started")
record("connected")

let dump = log["dump"]
print(dump(0))  // [started, connected]
```

Two things in that listing are worth pointing at. The closures go into the map by index assignment rather than `.set()`, for the reason we covered in the strategy-pattern warning. And each one is pulled out into a named variable before being called, because Lattice has no chained call syntax—`log["record"]("started")` does not parse.

Accumulate at the Edges

An accumulator is state, and state is what functional style is trying to keep away from the middle of your program. Use one where the work genuinely is stateful—collecting, tallying, logging—and keep the transformation stages pure. A pipeline whose every stage quietly mutates something is harder to reason about than the loop it replaced.

27.4 Currying and Partial Application Patterns

Currying and partial application are techniques for breaking a multi-argument function into a chain of single-argument functions. This lets you "pre-fill" some arguments and pass the specialized function around.

27.4.1 curry() — One Argument at a Time

The `F.curry()` function transforms a 2- or 3-argument function into a chain of single-argument functions:

Listing 27.22: Currying a 2-argument function

```
import "lib/fn" as F

let add = F.curry(|a, b| { a + b })

// Now add(5) returns a new function that adds 5
let add_five = add(5)
print(add_five(3))   // 8
print(add_five(10))  // 15

// Each stage is a value you can name and reuse
let add_two = add(2)
print(add_two(3))    // 5
```

Three-argument functions work the same way:

Listing 27.23: Currying a 3-argument function

```
import "lib/fn" as F

let volume = F.curry(|length, width, height| {
    length * width * height
})

let slab = volume(10)        // fix the length
let narrow_slab = slab(2)    // fix the width
print(narrow_slab(5))        // 100

let three_by_four = volume(3)
let box = three_by_four(4)
print(box(5))                // 60
```

Currying

Currying transforms a function `f(a, b)` into a chain of one-argument calls. Each call with one argument returns a new function that "remembers" the provided argument and waits for the rest.

Lattice Has No Chained Call Syntax

In languages with curried functions you would expect to write `add(2)(3)` and be done. Lattice's grammar does not allow it—a call expression cannot be immediately called again. The parser stops at the second open parenthesis on every backend:

```
error: 3:13: parse error: expected ',', got '('
```

The same restriction applies to any call-returning-callable, including `strategies.get("upper")(text)`. Name the intermediate result and call it on the next line. It costs one binding and reads better anyway.

27.4.2 partial() — Bind Some Arguments Now

While currying requires you to provide arguments one at a time in order, `F.partial()` lets you bind any number of leading arguments at once:

Listing 27.24: Partial application

```
import "lib/fn" as F

let greet = |greeting, name| { greeting + ", " + name + "!" }

let hello = F.partial(greet, "Hello")
let howdy = F.partial(greet, "Howdy")

print(hello("Alice"))  // Hello, Alice!
print(howdy("Bob"))    // Howdy, Bob!
```

You can bind multiple arguments at once:

Listing 27.25: Binding multiple arguments

```
import "lib/fn" as F

let log_message = |level, component, msg| {
    print("[" + level + "] " + component + ": " + msg)
}

let log_error = F.partial(log_message, "ERROR")
let log_db_error = F.partial(log_message, "ERROR", "database")

log_error("auth", "invalid token")
// [ERROR] auth: invalid token

log_db_error("connection timeout")
// [ERROR] database: connection timeout
```

27.4.3 Currying with Iterators

Currying and partial application become especially powerful when combined with iterators, because they let you create specialized transform functions on the fly:

Listing 27.26: Curried functions in pipelines

```
import "lib/fn" as F

// A curried multiplier
let multiply = F.curry(|factor, x| { factor * x })

let prices = [12.50, 8.99, 24.00, 5.75]

// Create specialized multipliers
let with_tax = multiply(1.08)
let discounted = multiply(0.85)

// Use them in pipelines
let taxed_prices = iter(prices).map(with_tax).collect()
print(taxed_prices)  // [13.5, 9.7092, 25.92, 6.21]

let sale_prices = iter(prices).map(discounted).collect()
print(sale_prices)  // [10.625, 7.6415, 20.4, 4.8875]
```

Compare this with the alternative—writing a fresh closure for each operation. Currying lets you express the intent ("multiply by 1.08") more directly than `|x| { x * 1.08 }`.

Curry or Partial?

Use `curry()` when you want to apply arguments one at a time and the function has 2–3 parameters. Use `partial()` when you want to bind one or more leading arguments in a single call, especially for functions with many parameters.

27.5 When Functional Style Shines in Lattice

Functional programming is a tool, not a religion. Lattice gives you both imperative and functional styles; the art is knowing when each one serves you best.

27.5.1 Data Transformation Pipelines

Functional style is at its best when you are transforming data through a series of well-defined steps:

Listing 27.27: A data transformation pipeline

```
import "lib/fn" as F

let raw_records = [
    "Alice,95,Math",
    "Bob,87,Science",
    "Carol,92,Math",
    "Dave,78,Science",
    "Eve,96,Math"
]

// Parse, filter, and summarize in one flow
let top_math_scores = pipe(raw_records,
    |records| { iter(records).map(|r| { r.split(",") }).collect() },
    |parsed| { iter(parsed).filter(|r| { r[2] == "Math" }).collect() },
    |math| { iter(math).map(|r| { to_int(r[1]) }).collect() },
    |scores| { iter(scores).filter(|s| { s >= 90 }).collect() }
)

print(top_math_scores)  // [95, 92, 96]
```

Each step has a clear purpose, and the whole pipeline reads like a recipe.

27.5.2 Callback-Heavy APIs

When working with APIs that accept callbacks—event handlers, sorting comparators, validation rules—functional tools help you build specialized callbacks without repetitive boilerplate:

Listing 27.28: Building validators with partial application

```
import "lib/fn" as F

let is_between = |min_val, max_val, x| {
    x >= min_val && x <= max_val
}

let is_valid_age = F.partial(is_between, 0, 150)
let is_valid_score = F.partial(is_between, 0, 100)

let ages = [25, -3, 42, 200, 17]
let valid_ages = iter(ages).filter(is_valid_age).collect()
print(valid_ages)  // [25, 42, 17]
```

Predicates like these are also your first line of defence against a callback that blows up mid-pipeline. A failure inside `.map()` or `.filter()` propagates out and stops the whole chain—see Section 26.5 for what that looks like and why guarding beats catching.

27.5.3 Configuration and Strategy Patterns

Functions are values in Lattice. You can store them in maps, pass them as arguments, and return them from other functions. This makes the strategy pattern natural:

Listing 27.29: Strategy pattern with functions

```
flux strategies = Map::new()
strategies["uppercase"] = |s| { s.to_upper() }
strategies["lowercase"] = |s| { s.to_lower() }
strategies["reverse"] = |s| { s.reverse() }

fn apply_strategy(strategy_name: String, text: String) -> String {
    let transform = strategies[strategy_name]
    return transform(text)
}

print(apply_strategy("uppercase", "hello"))   // HELLO
print(apply_strategy("reverse", "Lattice"))   // ecittaL
print(apply_strategy("lowercase", "SHOUT"))   // shout
```

Notice the index-assignment syntax. Storing a closure in a map is one of the few places where `m[k] = v` and `m.set(k, v)` are not interchangeable.

Storing Closures in Maps

As of v0.6.4, handing a *lambda literal* straight to `Map.set()` crashes the tree-walk evaluator:

```
let m = Map::new()
m.set("f", |s| { s + "!" })     // crashes under --tree-walk
```

Two spellings sidestep it, and both work on all three backends: index assignment (`m["f"] = |x| { ... }`, as above), or binding the closure to a name first and passing that name to `.set()`. The bytecode backends handle every form, so this only bites when you fall back to `--tree-walk` for debugging—which is exactly when you least want a crash.

27.5.4 When Imperative Style Is Better

Functional style is not always the clearest choice. Prefer imperative code when:

- The logic involves complex branching or early returns that don't fit neatly into a pipeline.
- You need to accumulate state across multiple collections simultaneously.
- The code would require deeply nested function calls that obscure the intent.
- Performance matters and you need fine-grained control over allocation.

Listing 27.30: Sometimes a loop is clearer

```
let items = [3, 18, 7, 42, 11, 25]
let threshold = 10

// Functional version -- hard to follow
let result = pipe(items,
    |xs| { iter(xs).filter(|x| { x > threshold }).collect() },
    |xs| {
        iter(xs).reduce(|acc, x| {
            if x > acc { x } else { acc }
        }, 0)
    }
)
print(result)   // 42

// Imperative version -- immediately clear
flux max_val = 0
for item in items {
    if item > threshold && item > max_val {
        max_val = item
    }
}
print(max_val)  // 42
```

The best Lattice code uses both styles where each is strongest. Functional pipelines for data transformations; imperative loops for stateful logic.

The Litmus Test

If you have to read a functional pipeline more than twice to understand what it does, it is probably trying to be too clever. Break it into named intermediate variables or switch to a loop. Clarity always beats cleverness.

27.6 Putting It All Together

Let's close with a larger example that combines iterators, `pipe()`, currying, and the `fn` module into a cohesive program.

Listing 27.31: A complete functional example

```
import "lib/fn" as F

// --- Data ---
let inventory = [
    ["Widget A", 29.99, 150],
    ["Widget B", 9.99, 500],
    ["Gadget C", 49.95, 30],
    ["Gadget D", 14.50, 200],
    ["Doohickey E", 4.99, 1000]
]

// --- Helpers built with currying ---
let min_price = F.curry(|threshold, item| { item[1] >= threshold })
let min_stock = F.curry(|threshold, item| { item[2] >= threshold })

// --- Pipeline: find valuable items with good stock ---
let report = pipe(inventory,
    |items| { iter(items).filter(min_price(10.0)).collect() },
    |items| { iter(items).filter(min_stock(100)).collect() },
    |items| {
        iter(items).map(|item| {
            let name = item[0]
            let total_value = item[1] * item[2]
            name + ": $" + to_string(total_value)
        }).collect()
    }
)

for line in report {
    print(line)
}
// Widget A: $4498.5
// Gadget D: $2900
```

Two rows survive. Widget B is plentiful but costs $9.99, so `min_price(10.0)` drops it; Gadget C is expensive but only 30 units deep, so `min_stock(100)` drops that one. (Gadget D's total lands on a whole number, and Lattice prints whole-number floats without a trailing `.0`—hence `$2900` rather than `$2900.0`.)

Each piece is small and testable on its own. The curried predicates can be reused across different pipelines. The `pipe()` call makes the overall flow readable. And because the iterator operations are lazy where possible, no unnecessary work is done.

27.7 Exercises

1. **Compose Three.** Lattice's built-in `compose()` takes exactly two functions. Write a function `compose3(f, g, h)` that composes three: `compose3(f, g, h)(x) = f(g(h(x)))`. Test it by composing three arithmetic operations.

2. **Curried Filter.** Using `F.curry()`, create a reusable `longer_than` predicate that filters strings by minimum length. Use it to extract all words longer than 5 characters from the sentence `"The quick brown fox jumped over the lazy dog"`.

3. **Word Frequency Pipeline.** Given a block of text (a multi-line string), use `pipe()` and the `fn` module to:

   (a) Split the text into words.
   (b) Convert all words to lowercase.
   (c) Count the frequency of each word using `F.frequencies()`.

   Print the resulting frequency map.

4. **Result Pipeline.** Write a sequence of operations using `F.ok()`, `F.flat_map_result()`, and `F.map_result()` to:

   (a) Start with a string that might represent a number.
   (b) Parse it to an integer (returning `F.err()` if it fails).
   (c) Double the number.
   (d) Check that it is positive (returning `F.err()` if not).

   Test with both valid and invalid inputs.

What's Next With iterators and functional tools in your kit, you are well-equipped to write expressive, composable Lattice code. In Chapter 28, we turn to a different kind of composition: organizing your code into *modules* and *packages*, so that the functions and types you build can be shared, reused, and managed as your projects grow.

Part VIII

Modules, Packages, and Project Structure

Chapter 28

Modules and Imports

As your programs grow beyond a single file, you face a fundamental organizational question: how do you split code into manageable pieces without losing the ability to share functionality between them? Most languages answer this with a module system, and Lattice is no different—except that Lattice's approach is deliberately minimal. There are no package declarations at the top of each file, no directory-based namespaces, no visibility modifiers beyond a single keyword. A file is a module. An `export` makes something public. An `import` brings it into scope.

Let's see how it works.

28.1 Your First Import

Suppose you have a utility file called `math_utils.lat` that defines a couple of functions:

Listing 28.1: math_utils.lat — a utility module

```
fn add(x: Int, y: Int) -> Int {
    return x + y
}

fn sub(x: Int, y: Int) -> Int {
    return x - y
}

let PI = 3.14159
```

Notice that this file has no `export` keywords. We will talk about what that means in a moment. For now, the important thing is that every top-level binding—`add`, `sub`, and `PI`—is visible to importers.

To use these from another file, you write:

Listing 28.2: main.lat — importing a module

```
import "math_utils" as math

print(math.add(2, 3))   // 5
print(math.sub(10, 4))  // 6
print(math.PI)           // 3.14159
```

The `import "math_utils" as math` statement does three things:

1. Locates the file `math_utils.lat` (appending the `.lat` extension automatically).
2. Executes that file's top-level code in an isolated scope.
3. Bundles the module's exported bindings into a map and assigns it to the name `math`.

You then access individual members with dot notation: `math.add`, `math.PI`, and so on.

File Extensions Are Optional

When you write `import "math_utils"`, Lattice automatically appends `.lat` if the path does not already end with it. You can also write `import "math_utils.lat"` explicitly—both forms resolve to the same file.

28.2 Import Styles

Lattice supports two import syntaxes, each suited to different situations.

28.2.1 Aliased Import: `import ... as`

The form we already saw brings in the entire module under a name you choose:

Listing 28.3: Aliased import

```
import "geometry/shapes" as shapes

let circle = shapes.make_circle(5.0)
let area = shapes.circle_area(circle)
```

This is the preferred style when you use many things from a module, or when you want the reader to see *where* each symbol comes from. The alias can be anything you like—`import "geometry/shapes" as geo` works too.

28.2.2 Selective Import: `import {...} from`

When you only need a few specific symbols, you can pull them directly into your local scope:

Listing 28.4: Selective import

```
import { add, PI } from "math_utils"

print(add(2, 3))  // 5
print(PI)         // 3.14159
```

Each name listed inside the braces becomes a local binding. You do not prefix them with a module name—`add` is `add`, not `math.add`.

This style is great when you need a handful of specific functions and want to keep your code concise. The trade-off is that the reader cannot immediately tell which module a function came from without scrolling back to the import.

Missing Exports Are Runtime Errors

If you try to selectively import a name that the module does not export, Lattice throws an error at runtime. For example, if `math_utils.lat` does not define `multiply`, then `import { multiply } from "math_utils"` will fail with:

```
module 'math_utils' does not export 'multiply'
```

The compiler generates a runtime check for each selectively imported name, validating that the field exists in the module map before binding it.
All three backends agree on that message; only the prefix differs. Run the program on the default VM and you see `vm error:` in front of it, on `--regvm` you see `regvm error:`, and on `--tree-walk` you see a bare `error:`. We will quote the core message from here on and leave the prefix implied.

Duplicate Selective Names Shadow Silently

Two selective imports that bring in the same name do not collide—the later one quietly replaces the earlier one, on every backend:

```
import { read_sensor } from "sensors/humidity"
import { read_sensor } from "sensors/pressure"

print(read_sensor())   // 1013.2 --- the pressure sensor wins
```

Nothing warns you. If two modules genuinely export the same name and you need both, reach for aliased imports instead (Section 28.6.4).

28.2.3 Bare Import

There is a third form you will see less often: a bare import with no alias and no selective names.

Listing 28.5: Bare import — side effects only

```
import "setup"
```

This executes the module's top-level code but discards the resulting module map. It is useful for modules that exist purely for their side effects—initializing a global configuration, registering plugins, or running setup code.

Bare Imports Are a Bytecode-VM Feature

The bare form works on the default stack VM and on `--regvm`. The tree-walking interpreter runs the module's top-level code and *then* refuses the statement:

```
error: import requires 'as <name>' or selective '{ ... } from'
```

If you need a module's side effects and want the program to run on every backend, bind the map and ignore it: `import "setup" as _setup`.

Module

A *module* in Lattice is a single `.lat` source file. When imported, the file is compiled and executed in an isolated scope. The module's top-level *value* bindings—functions, constants, variables, enums—are collected into a map that the importer can access. Struct declarations are compile-time entities and do not appear in that map (Section 28.3.2). There is no separate "module declaration" syntax—the file *is* the module.

28.3 The Export System

So far, our example modules have had no `export` keywords. In that case, Lattice uses *legacy mode*: every top-level binding is exported. This is convenient for small scripts and quick experiments, but for real projects you want explicit control over your module's public API.

28.3.1 Explicit Exports

Add the `export` keyword before any top-level declaration to mark it as public:

Listing 28.6: export_explicit.lat — explicit exports

```
export fn add(a: Int, b: Int) -> Int {
    return a + b
}

export let PI = 3.14159

fn internal_helper() -> Int {
    42
}

let secret = "hidden"
```

In this module, only `add` and `PI` are visible to importers. The function `internal_helper` and the value `secret` are private—they exist inside the module for its own use, but they do not appear in the module map.

Reaching for one anyway is not loud on every backend. Under the default VM and `--regvm`, `m.secret` evaluates to `nil`; under `--tree-walk` it raises `map has no key 'secret'`. A `nil` that silently propagates is the harder bug of the two, so when you are hunting a mysterious `nil` coming out of a module, check the export list first.

Legacy vs. Explicit Export Mode

When the parser encounters at least one `export` keyword in a file, it switches to *explicit mode*: only names marked with `export` are included in the module map. If a file contains *no* `export` keywords at all, it operates in *legacy mode*, where all top-level bindings are exported.
This two-mode system lets quick scripts "work" without ceremony, while larger projects get proper encapsulation.

The `export` keyword can precede:

- Functions: `export fn calculate(...)`
- Variables: `export let MAX_SIZE = 256` or `export fix THRESHOLD = 100`
- Structs: `export struct Point { ... }`
- Enums: `export enum Color { ... }`
- Traits: `export trait Drawable { ... }`

All five parse and load on every backend. Only the first three—functions, variables, and enums—actually put an entry into the module map; struct and trait declarations do not, which is what the next section is about.

28.3.2 Exporting Structs and Enums

Structs and enums travel across a module boundary by different routes, and the difference matters.

Listing 28.7: geometry.lat — exporting a struct behind a constructor

```
export struct Point {
    x: Int,
    y: Int
}

export fn make_point(x: Int, y: Int) -> Point {
    return Point { x: x, y: y }
}

export enum Direction { North, South, East, West }
```

An enum name can be imported like any other export, and the importer can then name its variants:

Listing 28.8: Using an exported enum and constructor

```
import { make_point, Direction } from "geometry"

let p = make_point(3, 4)
print(p.y)               // 4
print(Direction::South)  // Direction::South
```

A *struct type* is a different animal. Struct declarations are compile-time entities, not values, so they never land in the module map—and neither bytecode VM will let you import one by name:

```
import { Point } from "geometry"
// module 'geometry' does not export 'Point'
```

Even with an aliased import, `geo.Point` evaluates to `nil` and a bare `Point { x: 7, y: 8 }` in the importing file raises `unknown struct 'Point'`.

Hand Out Constructors, Not Types

Export a constructor function (and, if you need them, accessor or "with" functions) rather than the struct type itself. The importer never names `Point`; it calls `make_point` and works with the value it gets back, field access and all. This is the pattern that runs identically on all three backends, and it happens to be good encapsulation besides—the module keeps the freedom to change its representation later.

The Tree-Walker Is More Permissive Here

Under `--tree-walk`, importing a module also folds its struct declarations into the importer's struct table, so `import { Point } from "geometry"` succeeds and a bare `Point { ... }` literal constructs. Code written against that behavior breaks the moment it runs on the default VM. Treat the bytecode VMs as the language rule and use the constructor pattern.

Qualifying an enum through its alias works everywhere too—`geo.Direction::North` resolves at compile time. What does *not* work is naming a variant of an enum you never brought into scope: `Direction::East` with only an aliased import raises `undefined variable 'Direction::East'`.

28.3.3 What Gets Filtered Out

Regardless of export mode, Lattice always hides two categories of names:

1. Names starting with double underscores (`__`)—these are internal metadata used by the runtime.

2. Names containing a colon (`:`)—these are namespaced internal keys used for struct method dispatch and trait implementations.

You can see this filtering logic in `src/ast.c`, in the `module_should_export` function. It runs on every name in the module scope before adding it to the module map.

28.3.4 Re-exporting from Other Modules

A common pattern in larger projects is to create a "facade" module that imports from several internal modules and re-exports a curated public API:

Listing 28.9: api.lat — re-exporting a clean interface

```
import { double, BASE_CONST } from "internal/base_utils"

export fn double_it(x: Int) -> Int {
    return double(x)
}

export let REEXPORTED_CONST = BASE_CONST
```

The consumer of `api.lat` sees only `double_it` and `REEXPORTED_CONST`. They never need to know about `internal/base_utils.lat` at all.

28.4 Built-in Modules

Lattice ships with a set of built-in modules that are always available, regardless of where your script lives or whether you have any `lat_modules/` directory. These are implemented directly in the runtime (you can find the registration logic in `src/runtime.c`) and are loaded without touching the filesystem.

Here are the built-in modules:

To use a built-in module, import it by its bare name:

Table 28.1: Lattice's built-in modules

Module	Description
`math`	Mathematical functions and constants
`fs`	File system operations (read, write, stat, mkdir)
`path`	Path manipulation (join, basename, dirname, extension)
`json`	JSON parsing and serialization
`toml`	TOML parsing and serialization
`yaml`	YAML parsing and serialization
`crypto`	Cryptographic hashing (SHA-256, MD5, Base64)
`http`	HTTP client (GET, POST, PUT, DELETE)
`net`	Low-level networking (TCP sockets, DNS)
`os`	Operating system interface (env vars, exec, platform)
`time`	Time and date functions
`regex`	Regular expression matching
`progress`	Terminal progress bars

Listing 28.10: Using built-in modules

```
import "math" as math
import "json" as json
import "fs" as fs

let angle = math.PI() / 4.0
print(math.sin(angle))  // 0.707107

let data = json.parse("{\"name\": \"Lattice\"}")
print(data["name"])  // Lattice

let contents = fs.read_file("config.toml")
print(contents)
```

You can also use the selective import form with built-in modules:

Listing 28.11: Selective import from built-in modules

```
import { sin, cos, PI } from "math"

let x = cos(PI() / 3.0)
let y = sin(PI() / 3.0)
print("x=${x}, y=${y}")   // x=0.5, y=0.866025
```

`math.PI` Is a Function, Not a Constant

Every member of a built-in module is a native function, and the mathematical constants are no exception: `PI` and `E` are registered as zero-argument natives (you can see the two `mod_set_native` calls at the bottom of `build_math_module` in `src/runtime.c`). Write `math.PI()`, with the parentheses.

Forgetting them is worse than a typo, because the backends disagree about what happens next. On `--tree-walk` you get `unsupported binary operation on Closure and Float`; on the default VM and `--regvm` the arithmetic silently produces `0` and your circle collapses to a point.

Module Members Are Not Always Named What You Expect

The `fs` module reads files with `read_file`, not `read`. Ask for the wrong one and the error tells you exactly what happened, in three dialects: `type 'Map' has no method 'read'` (default VM), `no method 'read' on Map` (`--regvm`), `unknown method '.read()' on Map` (`--tree-walk`). A module map is an ordinary map, so a missing member is a missing key.

How Built-in Modules Are Detected

When the VM encounters an `import` statement, it first checks whether the path is a bare name (no directory separators, no leading dot). If so, it calls `rt_try_builtin_import()` to look up the name in a table of built-in module factories. If the name matches, the module map is constructed in C and returned directly—no file is read from disk. Only if this check fails does Lattice proceed to file-based module resolution. You can trace this logic in `src/stackvm.c`, in the `OP_IMPORT` handler.

28.5 Module Resolution

When you write `import "something"`, how does Lattice find the right file? The resolution algorithm follows a predictable sequence. Understanding it will save you from mysterious “module not found” errors.

28.5.1 The Resolution Order

For a given import path, Lattice tries these strategies in order:

1. **Built-in modules.** If the path is a bare name (like `"math"` or `"json"`), check the built-in module table. If found, return immediately.

2. **Package resolution.** If the path is still a bare name and step 1 did not match, try the `lat_modules/` directory. This looks for the module in several locations (detailed below).

3. **File path.** Treat the path as a file path. Append `.lat` if the path does not already end with it. Resolve to an absolute path using `realpath()`.

That third step hides the single most surprising rule in the module system, so let's pull it into the open.

28.5.2 Where "Relative" Is Relative To

On the bytecode VMs—the default backend and `--regvm`—a file-path import is resolved against the *process working directory*, not against the directory of the file that contains the import statement. There is no script-directory fallback. Watch what that means for a project laid out as `mods/main.lat` importing `mods/math_utils.lat`:

```
$ cd mods && clat main.lat
5
6
3.14159

$ cd .. && clat mods/main.lat
vm error: import: cannot find 'math_utils.lat'
```

Same program, same files, different working directory—and only one of them runs. You can trace the rule in the `OP_IMPORT` handler in `src/stackvm.c`: the path is handed straight to `realpath()`, which interprets a relative path against the CWD. Only the `lat_modules/` lookup in step 2 ever consults the script's directory.

The Tree-Walker Resolves Differently

Under `--tree-walk`, that second command succeeds—the tree-walking interpreter resolves imports relative to the importing script's directory. This is the largest behavioral gap between the backends in the whole module system, and it bites in exactly the way you would expect: a project that works during a debugging session with `--tree-walk` fails under the default VM in CI.
Write your programs to the bytecode-VM rule, which is the stricter of the two.

Living With CWD-Relative Imports

Three habits keep this from ever hurting:

- **Run from the project root.** Make the root the working directory and write every import relative to it. This is what the `lattice.toml` layout in Section 28.6 is for.
- **Use `lat_modules/` for anything shared.** Package resolution (step 2) tries the script's directory first, so bare-name imports are position-independent in a way that path imports are not.
- **Use absolute paths in generated code.** Scaffolding tools that emit Lattice source—project generators, build scripts—should write absolute import paths rather than hoping the user runs `clat` from the right folder.

28.5.3 Package Resolution Strategies

When resolving a bare module name through `lat_modules/`, Lattice tries four paths in order (implemented in `pkg_resolve_module()` in `src/package.c`):

1. `lat_modules/⟨name⟩/main.lat` — a package with a standard entry point.
2. `lat_modules/⟨name⟩/src/main.lat` — a package using a `src/` subdirectory.
3. The `entry` field from `lat_modules/⟨name⟩/lattice.toml` — a package with a custom entry point defined in its manifest.
4. `lat_modules/⟨name⟩.lat` — a single-file package.

Here "project directory" means the directory of the script that `clat` was pointed at, which is why bare-name package imports keep working no matter where you run from. If none of the four hit, Lattice retries only two of them—`lat_modules/⟨name⟩/main.lat` and `lat_modules/⟨name⟩.lat`—relative to the current working directory. Strategies 2 and 3 are never retried against the CWD.

Listing 28.12: Importing a third-party package

```
// If you have lat_modules/leftpad/main.lat installed:
import "leftpad" as lp

let padded = lp.pad("hello", 10)
print(padded)  // "     hello"
```

An `entry` Cannot Escape Its Package

Strategy 3 reads a path out of a dependency's own `lattice.toml`, which makes it the one resolution step driven by data you did not write. Lattice confines the result: after resolving, the absolute path must still live inside `lat_modules/⟨name⟩/`. A manifest that asks for `entry = "../../../evil.lat"` resolves to nothing at all rather than to a file outside the package. A legitimate sub-path such as `entry = "src/main.lat"` resolves normally. The check lives in `pkg_resolve_module()` in `src/package.c`.

Native Extensions Use a Different Search Path

`import` loads Lattice source. Loading a compiled native extension is a separate mechanism, `require_ext()`, with its own search order and its own trust model (Chapter 35). It looks in `~/.lattice/ext/` first, then `$LATTICE_EXT_PATH`. A CWD-relative `./extensions/` directory is searched *only* when the environment variable `LATTICE_EXT_ALLOW_CWD` is set, and even then it is tried last—so that dropping a shared library into a project folder cannot silently hijack an extension name. When the lookup fails, the error names the directories it tried:

```
require_ext: cannot find extension 'nosuch'
(searched ~/.lattice/ext/, $LATTICE_EXT_PATH)
```

28.5.4 Relative and Nested Imports

For your own project files, use relative paths:

Listing 28.13: Relative path imports

```
// Import a file in the same directory
import "utils" as utils

// Import a file in a subdirectory
import "models/user" as user_model

// Import a file in a parent directory
import "../shared/config" as config
```

Every one of those is resolved against the working directory you launched `clat` from (Section 28.5.2), so "the same directory" means "the same directory as the one you are standing in". Absolute paths work too and sidestep the question entirely:

```
import "/opt/lattice-shared/config" as config
```

Use Consistent Project Structure

A good convention is to keep your project's main entry point at the root and organize supporting modules in subdirectories:

```
my_project/
  main.lat
  models/
    user.lat
    post.lat
  utils/
    validation.lat
    formatting.lat
  lattice.toml
```

Then your imports read naturally: `import "models/user" as user`, `import "utils/validation" as valid`.

28.5.5 Module Caching

Lattice caches modules by their *resolved absolute path*. If two different files both import `"utils"`, the module is compiled and executed only once. Subsequent imports receive a deep clone of the cached module map.

This has two important consequences:

1. **Side effects run once.** If a module's top-level code prints a message or opens a file, that happens during the first import. Later imports skip execution entirely and use the cached result.

2. **Each importer gets its own copy.** Because the cache returns a deep clone, modifying a value obtained from a module does not affect other importers. The phase system still applies—if you import a frozen value, you get a frozen copy.

Listing 28.14: counter.lat — a module that announces itself

```
print("counter module loaded!")

export let START = 0

export fn describe() -> String {
    return "counter starting at ${START}"
}
```

Now import it from two intermediate modules *and* from the program itself—three importers, one load:

Listing 28.15: alpha.lat and beta.lat both pull in counter.lat

```
// alpha.lat
import "counter" as c
export fn alpha_report() -> String { return "alpha: " + c.describe() }

// beta.lat
import "counter" as c
export fn beta_report() -> String { return "beta: " + c.describe() }
```

Listing 28.16: main.lat — the banner appears exactly once

```
import "alpha" as a
import "beta" as b
import "counter" as c

print(a.alpha_report())
print(b.beta_report())
print(c.describe())

// counter module loaded!
// alpha: counter starting at 0
// beta: counter starting at 0
// counter starting at 0
```

The banner prints once, before any of the reports, and identically on all three backends.

To see the second consequence—the deep clone—have two importers reach into the same exported array:

Listing 28.17: Each importer gets a private copy

```
// shared.lat
export flux log = []

// clone_test.lat
import "shared" as s1
import "shared" as s2

s1["log"].push("from s1")
print(s1["log"].len())  // 1
print(s2["log"].len())  // 0
```

The push lands in s1's copy and nowhere else. A module is not a shared mutable singleton; it is a template that each importer receives its own casting of.

Mutate Module Fields Through Index Syntax

Notice the `s1["log"]` in that listing. Dot access on a map yields a *copy* of the field, so `s1.log.push("x")` runs without complaint and mutates nothing—the length stays `0` on every backend. Index syntax resolves to the live element and the mutation sticks. This is a property of maps generally, not of modules, but module maps are where it tends to be discovered.

28.6 Organizing Multi-File Projects

Now that we understand the mechanics, let's look at how to organize a real project.

28.6.1 A Minimal Project

The smallest structured Lattice project looks like this:

```
my_app/
  lattice.toml
  main.lat
```

The `lattice.toml` file declares your project's metadata and dependencies (we will cover it in detail in the next chapter). The `main.lat` file is the entry point.

28.6.2 Growing Beyond One File

As the project grows, extract cohesive functionality into modules:

```
weather_station/
  lattice.toml
  main.lat
  sensors/
    temperature.lat
    humidity.lat
    pressure.lat
  alerts/
    thresholds.lat
    notifier.lat
  utils/
    formatting.lat
    logging.lat
```

Listing 28.18: weather_station/main.lat — tying it all together

```
import "sensors/temperature" as temp
import "sensors/humidity" as humid
import "alerts/thresholds" as thresh
import "alerts/notifier" as notify
import "utils/formatting" as fmt

let reading = temp.read_temperature()
let humidity = humid.read_humidity()

if reading > thresh.HIGH_TEMP {
    notify.send_alert(fmt.format_reading(reading, "F"))
}

print(fmt.format_reading(reading, "F"))   // 72.4 F
print(fmt.format_reading(humidity, "%"))  // 41 %
```

Listing 28.19: sensors/temperature.lat

```
export fn read_temperature() -> Float {
    // In a real application, this would read from hardware
    return 72.4
}

export fn to_celsius(fahrenheit: Float) -> Float {
    return (fahrenheit - 32.0) * 5.0 / 9.0
}
```

Run this from the `weather_station/` directory—the import paths are resolved against your working directory, not against `main.lat`'s location.

Listing 28.20: alerts/thresholds.lat

```
export let HIGH_TEMP = 100.0
export let LOW_TEMP = 32.0
export let HIGH_HUMIDITY = 80.0
export let LOW_HUMIDITY = 20.0
```

28.6.3 Facade Modules

When a subsystem has many internal modules, create a facade that presents a clean interface. A facade is a module that imports other modules, and that combination comes with a rule we have to state before writing any code: **a module that is itself imported must use selective imports, not aliased ones.** We will see why in a moment. First, the facade:

Listing 28.21: sensors/mod.lat — a facade module

```
import { read_temperature, to_celsius } from "sensors/temperature"
import { read_humidity } from "sensors/humidity"
import { read_pressure } from "sensors/pressure"

// Re-export a curated API
export fn temperature_f() -> Float {
    return read_temperature()
}

export fn temperature_c() -> Float {
    return to_celsius(read_temperature())
}

export fn humidity_pct() -> Float {
    return read_humidity()
}

export fn pressure_hpa() -> Float {
    return read_pressure()
}
```

Listing 28.22: A consumer sees only the curated names

```
import "sensors/mod" as sensors

print(sensors.temperature_f())   // 72.4
print(sensors.temperature_c())   // 22.4444
print(sensors.humidity_pct())    // 41
print(sensors.pressure_hpa())    // 1013.2
```

Because selective imports share one flat namespace, the three sensor modules have to export *distinct* names—`read_temperature`, `read_humidity`, `read_pressure`—rather than three functions all called `read_sensor`. That is a small design tax, and it buys a facade that runs identically on all three backends.

Aliased Imports Do Not Work Inside an Imported Module

On the default stack VM, an alias bound by `import ... as` inside a module that is itself imported cannot be used—not from the module's exported functions, and not even from its own top-level code.

```
// mid.lat
import "leaf" as leaf
export fn relay() -> Int { return leaf.value() }

// top.lat
import "mid" as mid
print(mid.relay())
```

```
vm error: undefined variable 'leaf' (did you mean 'lerp'?)
```

The parenthetical is the bytecode VMs' habit of appending the nearest global builtin to an undefined-variable error; `lerp` is a real math builtin, and it is pure noise here. If the alias is used at the module's top level instead, the module load itself fails and the importer receives `nil`, which surfaces one step later as `type 'Nil' has no method 'relay'` (the default VM's phrasing for a missing method; Section 2.5.7 lists the other two). Binding an alias you never use is harmless; touching it is not.

`--tree-walk` and `--regvm` both handle this case correctly, which is exactly what makes it dangerous—the facade works everywhere except the default backend. Selective imports and bare imports inside a module have no such problem. Aliased imports in your top-level script, the ones you write every day, are also completely fine; the restriction applies only one level down.

If you truly need an alias inside a module—say two dependencies really do export the same name—move the import into the function body, where it compiles to a local rather than a module-scope global:

```
// mid.lat --- works on all three backends
export fn relay() -> Int {
    import "leaf" as leaf
    return leaf.value()
}
```

No Implicit Index Files

Unlike Node.js, Lattice does not automatically look for an `index.lat` or `mod.lat` when you import a directory name. If you write `import "sensors"`, Lattice looks for `sensors.lat`—and when it does not find one, says so on every backend:

```
import: cannot find 'sensors.lat'
```

To use a facade, import it explicitly: `import "sensors/mod" as sensors`.

28.6.4 Namespacing with Aliased Imports

When two modules export the same name, aliased imports prevent collisions:

Listing 28.23: Avoiding name collisions

```
import "models/user" as user_mod
import "models/admin" as admin_mod

// Both modules export make_record, but there is no conflict
let regular = user_mod.make_record("alice", "alice@example.com")
let admin = admin_mod.make_record("bob", "admin")
```

Selective imports would not report a conflict here—they would quietly pick one. Both `make_record` bindings compete for the same name and the later import wins, silently, with no diagnostic on any backend (Section 28.2.2). An aliased import keeps the two apart by construction, and it does so in the one place selective imports cannot: a top-level script.

Choosing Between the Two Forms

Aliased imports for scripts and entry points, where you want the reader to see which module every call came from. Selective imports inside modules that will themselves be imported, where the aliased form does not work on the default backend anyway. That single rule of thumb resolves nearly every case.

28.6.5 Scoped Imports

Import statements are not limited to the top of a file. You can import inside a function body, which makes the module available only within that scope:

Listing 28.24: Scoped import inside a function

```
fn stringify_pair(a: Int, b: Int) -> String {
    import "json" as json

    flux m = Map::new()
    m["a"] = a
    m["b"] = b
    return json.stringify(m)
}

print(stringify_pair(1, 2))  // {"b":2,"a":1}
print(json)                  // undefined variable 'json'
```

The last line is the point: outside the function, the name is gone. (On the bytecode VMs the message carries a "did you mean" suggestion drawn from the global builtins—`undefined variable 'json' (did you mean 'asin'?)`—while `--tree-walk` prints the bare `undefined variable 'json'`. The core message is the same everywhere.)

Scoped imports are compiled to local variable definitions rather than globals. The module is still cached globally, so there is no performance penalty for importing inside a function that gets called many times.

28.7 Under the Hood: How Imports Work

For readers who want to understand the machinery, here is what happens when Lattice encounters an import statement.

28.7.1 Parsing

The parser (in `src/parser.c`) recognizes the syntactic forms and creates a `STMT_IMPORT` AST node. The node stores four fields:

- `module_path` — the string literal from the import (always present).
- `alias` — the identifier after `as` (null if not provided).
- `selective_names` — an array of identifiers from the `{...}` list (null for aliased or bare imports).
- `selective_count` — how many names that array holds.

A bare import is the shape with all of `alias`, `selective_names`, and `selective_count` empty.

28.7.2 Compilation

The compiler (`src/stackcompiler.c`) turns the AST node into bytecode. The core instruction is `OP_IMPORT`, which takes a single byte-wide operand: the index of the module path in the chunk's constant pool.

For an aliased import like `import "utils" as u`:

1. Emit `OP_IMPORT` with the path constant index.
2. Emit `OP_DEFINE_GLOBAL` — or its wide sibling `OP_DEFINE_GLOBAL_16` when the alias constant sits past index 255 — with the alias name. Inside a function, add a local instead.

For a selective import like `import { add, PI } from "math_utils"`, the compiler generates a more elaborate sequence for each name:

1. Emit `OP_DUP` to copy the module map on the stack (or `OP_GET_LOCAL` when the import is inside a function, where the map lives in a slot).
2. Emit `OP_GET_FIELD` to extract the named export.
3. Emit `OP_DUP` and `OP_JUMP_IF_NOT_NIL` to validate the export exists.
4. If nil, emit `OP_THROW` with an error message like `"module 'math_utils' does not export 'multiply'"` — the string is baked into the constant pool at compile time, which is why the message can name both the module and the missing export.
5. Otherwise, emit `OP_DEFINE_GLOBAL` (or add local) for that name.

After processing all selective names, a final `OP_POP` discards the module map from the stack.

Why the Import Path Operand Is Only a Byte Wide

Constant pushes, global reads and writes, and closure creation all have 16-bit variants for chunks with more than 256 constants (Section 34.2). `OP_IMPORT` does not. Put an import after a few hundred string literals in the same chunk and the compiler stops rather than emitting a truncated operand:

```
compile error: too many constants in one chunk for a
field/method/name operand (>255)
```

`--regvm` refuses with its own phrasing and a suggestion: `too many constants in one chunk for an 8-bit name operand (split the function or reduce literals)`. It is a loud failure rather than a silent miscompile, and in practice the fix is the one the register VM suggests.

28.7.3 Runtime Execution

When the VM hits `OP_IMPORT` at runtime (in `src/stackvm.c`), it:

1. **Reads the path** from the constant pool.

2. **Checks built-in modules** via `rt_try_builtin_import()`. If the path matches a stdlib module, the C implementation constructs a module map and pushes it onto the stack. Done.

3. **Resolves the file path.** Tries `pkg_resolve_module()` for bare names (checking `lat_modules/` relative to the script's directory, then the working directory), and otherwise appends `.lat` and hands the result to `realpath()`. That last call is where the working-directory rule of Section 28.5.2 comes from: `realpath()` interprets a relative path against the CWD, and the bytecode VM never consults the script's own directory for path imports.

4. **Checks the cache.** The VM maintains a hash map (`vm->module_cache`) keyed by absolute path. If the module has been loaded before, a deep clone of the cached module map is pushed and execution continues.

5. **Reads and compiles the source file.** The source is lexed, parsed, and compiled into a module chunk. Module chunks are compiled without an auto-call to `main()`.

6. **Executes in an isolated scope.** The VM pushes a new scope with `env_push_scope()`, runs the module's bytecode, then pops the scope. This ensures the module's internal variables do not leak into the caller's environment.

7. **Builds the module map.** After execution, the VM iterates over the module scope's bindings. Each binding passes through `module_should_export()` to determine visibility. Exported bindings are deep-cloned into a new map.

8. **Caches and returns.** The module map is stored in the cache and pushed onto the stack for the caller to use.

Closures Capture the Module Environment

When a module exports a function, that function is a closure that captures the module's environment. Exported functions can therefore call other functions defined in the same module—even ones that are not exported—and read the module's own top-level constants:

```
// privhelper.lat
fn secret_double(n: Int) -> Int { return n * 2 }
let SCALE = 10
export fn compute(n: Int) -> Int { return secret_double(n) + SCALE }

// caller: p.compute(5) --> 20 on all three backends
```

The VM copies module bindings into the base scope before popping the module's scope, so those closure environments stay valid after the import returns. The gap in that copy is the aliased-import binding, which is why the limitation described in Section 28.6.3 exists.

28.8 Exercises

1. **Split and import.** Take any program you have written so far that is longer than 50 lines. Extract at least two cohesive groups of functions into separate module files. Use explicit `export` to control what each module exposes. Verify that the program still works after the split.

2. **Selective vs. aliased.** Write a module called `colors.lat` that exports at least five functions (`to_hex`, `to_rgb`, `lighten`, `darken`, `mix`). Write two consumer files: one that uses an aliased import and another that uses selective imports. Which style do you prefer for this module? Why?

3. **The facade pattern.** Create a `database/` directory with three modules: `connection.lat`, `query.lat`, and `migration.lat`. Each should export at least two functions, with names that do not collide. Then create `database/mod.lat` that re-exports a curated subset of the combined API, using selective imports. Write a `main.lat` that imports only from the facade, and run it on all three backends.

4. **Built-in module explorer.** Write a program that imports the `math`, `json`, and `os` built-in modules. Use `math.sin` and `math.cos` to compute four points on a unit circle, serialize them to JSON with `json.stringify`, and print the current platform with `os.platform()`. Remember that `math.PI` is a function call. Now deliberately drop the parentheses and run the result on each backend—you should get an error on one and a wrong answer on two.

5. **Module caching investigation.** Create a module called `counter.lat` that prints `"loaded!"` at the top level and exports a function. Import it from two other modules *and* from your entry point, all in the same program. Verify that `"loaded!"` is printed only once. Then modify one of the importers to use the module inside a function body (scoped import). Does the caching behavior change?

6. **Where does "relative" point?** Build the two-file project from Section 28.5.2: `mods/main.lat` importing `mods/math_utils.lat`. Run it from inside `mods/`, then from the parent directory, on each of the three backends. Record which combinations work. Then pre-compile `mods/main.lat` with `clat compile` and repeat the experiment on the resulting `.latc` file (Chapter 34)—does shipping bytecode change the answer?

What's Next

Now that we can split code across files and control what each module exposes, the next question is: how do we manage *other people's* code? In Chapter 29, we will explore Lattice's built-in package manager—`clat init`, `lattice.toml`, semantic versioning, dependency resolution, and the `lat_modules/` directory that we've already been referencing. The module system you just learned is the foundation; the package manager builds the ecosystem on top of it.

Chapter 29

The Package Manager

In the previous chapter, we learned how to split our own code across modules and import them. But programming does not happen in isolation. Eventually, you will want to use a library someone else wrote—a JSON schema validator, a CLI argument parser, a color formatting toolkit. That is where the package manager comes in.

Lattice's package manager is built directly into `clat`, the same binary you use to run programs. There is no separate tool to install, no Node.js-style global package manager that updates independently from the language. Everything lives in one place: your manifest, your lock file, and a `lat_modules/` directory that works much like you would expect if you have used `npm` or `cargo`.

Let's start by creating a project.

29.1 Initializing a Project with `clat init`

Every managed Lattice project begins with a `lattice.toml` file. You can create one by hand, but the fastest way is:

```
mkdir weather-app && cd weather-app
clat init
```

Output:

```
Created lattice.toml
```

Open the generated file and you will see:

Listing 29.1: A freshly generated lattice.toml

```
[package]
name = "weather-app"
version = "0.1.0"
```

Three things to notice:

1. The `name` is derived from the current directory name.
2. The `version` starts at `0.1.0`, following semantic versioning conventions.
3. There is no explicit `entry` field—when omitted, Lattice defaults to `main.lat` as the entry point.

If you run `clat init` in a directory that already has a `lattice.toml`, it refuses and prints an error. It will never overwrite an existing manifest.

Custom Entry Points

If your project's main file is not called `main.lat`, add an `entry` field to the `[package]` section:

```
[package]
name = "weather-app"
version = "0.1.0"
entry = "src/app.lat"
```

This tells both the package manager and importers where to find the project's primary module.

29.2 The `lattice.toml` Manifest

The `lattice.toml` file is the single source of truth for your project's metadata and dependencies. It uses TOML syntax—the same format Rust uses for `Cargo.toml` and Python uses for `pyproject.toml`.

Here is a complete manifest for a project with dependencies:

Listing 29.2: A lattice.toml with metadata and dependencies

```
[package]
name = "weather-app"
version = "1.2.0"
description = "A weather monitoring application"
license = "MIT"
entry = "main.lat"

[dependencies]
http-client = "^1.0.0"
json-schema = "~0.3.2"
cli-args = "2.1.0"
color-fmt = "*"
```

29.2.1 The [package] Section

The [package] table contains your project's metadata:

Table 29.1: Fields in the [package] section

Field	Required	Description
name	Yes	The package name (derived from directory by clat init)
version	Yes	Semantic version string (e.g., "1.2.3")
description	No	A short description of the package
license	No	SPDX license identifier (e.g., "MIT", "Apache-2.0")
entry	No	Entry point file (defaults to "main.lat")

29.2.2 The [dependencies] Section

The [dependencies] table maps package names to version constraints. Each key is a package name, and each value is a string specifying which versions are acceptable:

```
[dependencies]
http-client = "^1.0.0"
json-schema = "~0.3.2"
cli-args = "2.1.0"
color-fmt = "*"
```

We will break down what those version strings mean in Section 29.4.

29.3 Adding, Removing, and Installing Packages

The package manager provides three commands for managing dependencies.

29.3.1 `clat add`

To add a dependency to your project:

```
clat add http-client ^1.0.0
```

Output:

```
Added http-client@^1.0.0 to dependencies
  Installing http-client@^1.0.0... ok

All dependencies installed.
```

This does two things:

1. Adds `http-client = "^1.0.0"` to the `[dependencies]` section of your `lattice.toml`.
2. Runs `clat install` to fetch and install the package.

If you omit the version constraint, it defaults to `"*"` (any version):

```
clat add color-fmt
```

If the package is already a dependency, `clat add` updates its version constraint instead of adding a duplicate.

No `lattice.toml` Yet?

If you run `clat add` before running `clat init`, the package manager creates a minimal `lattice.toml` for you automatically, deriving the package name from the current directory. You do not need a separate initialization step if you start by adding dependencies.

29.3.2 Git Dependencies

Not every package lives in a registry. Sometimes the library you need is a repository on GitHub, or an internal repo behind your company's VPN, or a fork you maintain yourself. Since v0.3.29, `clat add` can pull a dependency straight from git:

```
clat add colorfmt --git https://github.com/example/colorfmt --tag v0.4.0
```

Three ref selectors are available, and they are mutually exclusive:

Table 29.2: Git ref selectors for `clat add`

Flag	Meaning
`--tag ⟨tag⟩`	Check out a specific tag (the reproducible choice)
`--branch ⟨branch⟩`	Track a branch tip
`--rev ⟨commit⟩`	Pin an exact commit hash

The command writes an inline table into your manifest rather than a bare version string:

Listing 29.3: A git dependency in lattice.toml

```
[package]
name = "gituser"
version = "0.1.0"

[dependencies]
colorfmt = { git = "https://github.com/example/colorfmt", tag = "v0.4.0" }
```

You can also write that table by hand—the `git`, `tag`, `branch`, `rev`, and `version` keys are all recognized by the manifest parser.

Behind the scenes, the package manager clones the repository into the global cache at `~/.lattice/packages/⟨name⟩/git-⟨ref⟩/`, checks out the requested ref, then copies the working tree into `lat_modules/` *with the* `.git` *directory stripped*. Your `lat_modules/` stays a plain directory of source files—no nested repositories, no submodule surprises.

Listing 29.4: Installing a git dependency

```
Added colorfmt from git (https://github.com/example/colorfmt)
  Installing colorfmt@v0.4.0...
  ... git checkout output ...
  colorfmt (git v0.4.0)
 ok

All dependencies installed.
```

Git's own chatter (the "detached HEAD" advisory, for instance) passes through to your terminal during the checkout. That is git talking, not `clat`.

Prefer Tags and Revisions Over Branches

A `-branch` dependency resolves to whatever the branch tip happens to be the day you install it. A `-tag` or `-rev` dependency resolves to the same tree forever. Since the lock file records a content checksum (Section 29.5), a moving branch will make your lock file and your working tree disagree the moment upstream pushes.

29.3.3 `clat install`

To install all dependencies listed in your manifest:

```
clat install
```

If there is no manifest in the current directory, the installer says so and points you at `clat init`:

```
error: no lattice.toml found. Run 'clat init' first.
```

Otherwise:

Output:

```
  Installing http-client@^1.0.0... ok
  Installing json-schema@~0.3.2... ok
  Installing cli-args@2.1.0... ok
  Installing color-fmt@*... ok

All dependencies installed.
```

For each dependency, the installer follows this resolution strategy:

1. **Check `lat_modules/`.** If the package directory already exists, verify that the installed version satisfies the constraint. If it does, skip the download.
2. **Check the lock file.** If `lattice.lock` exists, prefer the locked version (as long as it still satisfies the manifest constraint). This ensures reproducible builds.
3. **Clone from git.** If the dependency carries a `git` key, clone (or reuse a cached clone of) the repository at the requested ref (Section 29.3.2).
4. **Check the local file registry.** If the `LATTICE_REGISTRY` environment variable is set to a `file://` URL, try copying the package from that directory.
5. **Fetch from the HTTPS registry.** Query the registry for available versions, find the best match, download it to the global cache, and copy it into `lat_modules/`.

After all dependencies are installed, `clat install` builds a dependency graph and checks for circular dependencies. If a cycle is detected, it reports the full cycle path and fails:

```
error: circular dependency detected: alpha -> beta -> gamma -> alpha
```

Finally, it writes (or updates) the `lattice.lock` file.

29.3.4 `clat remove`

To remove a dependency:

```
clat remove color-fmt
```

Output:

```
Removed color-fmt
```

This does three things:

1. Removes the package from `lattice.toml`.
2. Deletes `lat_modules/color-fmt/` from disk.
3. Removes the entry from `lattice.lock`.

If the package is not listed as a dependency, `clat remove` prints an error and does nothing:

```
$ clat remove nothere
error: 'nothere' is not a dependency
```

29.4 Semantic Versioning and Dependency Resolution

Lattice uses semantic versioning (semver) for all version strings. A version has three numeric components:

Semantic Version

A version string `MAJOR.MINOR.PATCH` communicates intent:

- **MAJOR** — incremented for breaking, incompatible changes.
- **MINOR** — incremented for backward-compatible new features.
- **PATCH** — incremented for backward-compatible bug fixes.

For example, `2.3.1` means major version 2, minor version 3, patch level 1.

29.4.1 Version Constraints

When you list a dependency in `lattice.toml`, the version string is a *constraint*, not an exact version. Lattice supports six constraint types:

29.4.2 Caret vs. Tilde

The most commonly used constraints are caret (`^`) and tilde (`~`). Here is how they differ in practice:

Table 29.3: Semver constraint types

Syntax	Name	Meaning
`"*"`	Wildcard	Any version. Useful during early development.
`"1.2.3"`	Exact	Exactly version 1.2.3 and nothing else.
`"^1.2.3"`	Caret (Compatible)	Same major version, at least 1.2.3. Matches `1.2.3`, `1.3.0`, `1.99.0`, but not `2.0.0`.
`"~1.2.3"`	Tilde (Approximate)	Same major *and* minor version, at least patch 3. Matches `1.2.3`, `1.2.9`, but not `1.3.0`.
`">=1.2.3"`	Minimum	Version 1.2.3 or newer. No upper bound.
`"<=1.2.3"`	Maximum	Version 1.2.3 or older. No lower bound.

Listing 29.5: Caret vs. tilde constraints

```
# Caret: ^1.2.3
#   Allows: 1.2.3, 1.2.4, 1.3.0, 1.99.0
#   Blocks: 2.0.0 (different major)

# Tilde: ~1.2.3
#   Allows: 1.2.3, 1.2.4, 1.2.99
#   Blocks: 1.3.0 (different minor)
```

When to use caret: You trust the package author to maintain backward compatibility within a major version. This is the default choice for most dependencies.

When to use tilde: You want only bug fixes for a specific feature set. This is more conservative and useful when a minor version introduced a behavior you depend on.

Default to Caret

If you are unsure which constraint to use, start with caret (`^`). It gives package authors room to ship new features without breaking your code, while still protecting you from major-version breaking changes.

29.4.3 How Resolution Works

When `clat install` needs to resolve a version constraint, it follows this process:

1. **Query the registry** for all published versions of the package. The registry endpoint is `GET /packages/<name>/versions`, which returns a JSON array.

2. **Filter** the version list against the constraint using `pkg_semver_satisfies()` (implemented in `src/package.c`).
3. **Select the highest** matching version. Lattice always prefers the newest compatible version, following the common "maximal version" strategy.

For example, if the registry has versions `[1.0.0, 1.1.0, 1.2.0, 2.0.0]` and your constraint is `^1.0.0`, the resolver picks `1.2.0`—the highest version that shares the same major version.

29.5 The Lock File and Reproducible Builds

Every time `clat install` succeeds, it writes a `lattice.lock` file. This file records the *exact* versions that were installed *and a cryptographic fingerprint of what landed on disk*, so that future installs produce identical results.

29.5.1 What the Lock File Looks Like

Listing 29.6: A real lattice.lock, with checksums

```
# This file is auto-generated by clat. Do not edit manually.

[[package]]
name = "greetings"
version = "1.2.0"
source = "path"
checksum = "f52554b8f94eed7f4bc54ae9b7c73f47dfe1a5e01389da3756f566ce20fd0bf2"

[[package]]
name = "colorfmt"
version = "0.4.0"
source = "git"
checksum = "a812ac1353e698056cdfd5f7f4db75f416a2e542a166e9213bec0dcb298b535f"
```

Each `[[package]]` entry records:

- **`name`** — the package name.
- **`version`** — the exact resolved version.
- **`source`** — where the package came from: `"registry"` (HTTP registry), `"local"` (already in `lat_modules/`), `"path"` (file-based registry), or `"git"` (cloned repository).
- **`checksum`** — the SHA-256 of the installed package tree.

29.5.2 Checksums and Integrity Verification

As of v0.4.0, that `checksum` field is not decorative. After each package is placed in `lat_modules/`, the installer walks the package tree, hashes it with SHA-256, and compares the result against whatever the lock file already recorded. If the two disagree, the install *fails closed*:

Listing 29.7: An integrity check catching a modified package

```
$ clat install
  Installing greetings@1.2.0... FAILED
  error: integrity check failed for 'greetings' — content does not match lattice.lock
    checksum

1 package(s) failed to install.
```

The command exits with status 1 and the offending package is left alone rather than silently overwritten.

This catches three distinct things: a registry that served different bytes for the same version, a cache entry that was corrupted or tampered with, and—most often in practice—a teammate who edited a file inside `lat_modules/` to test a fix and forgot about it.

Legitimately Changed a Dependency?

If you *meant* to change the installed tree, delete the package's `[[package]]` entry from `lattice.lock` (or delete `lattice.lock` entirely) and run `clat install` again. The installer records the new checksum instead of comparing against the old one.

29.5.3 How the Lock File Ensures Reproducibility

When `clat install` finds an existing `lattice.lock`, it prefers the locked version for each dependency—*as long as it still satisfies the constraint in* `lattice.toml`. This means:

- **Same machine, next week:** Running `clat install` again installs the same versions, even if newer versions have been published.
- **Different machine, same lock file:** A teammate clones your repository, runs `clat install`, and gets the exact same dependency versions.
- **Updating the constraint:** If you change a version constraint in `lattice.toml` so that the locked version no longer satisfies it, the next `clat install` fetches a new version and updates the lock file.

Commit Your Lock File

Always commit `lattice.lock` to version control. It is how you guarantee reproducible builds across machines and over time—and, since v0.4.0, it is also the record your teammates' installs verify against. An uncommitted lock file means nobody downstream can tell whether they received the same bytes you did. The `lat_modules/` directory, on the other hand, should be in your `.gitignore`—it can always be reconstructed from the lock file.

29.5.4 When Lock Entries Are Updated

The lock file is regenerated in these situations:

1. `clat install` runs and resolves new or changed dependencies.
2. `clat add` adds a new dependency and triggers an install.
3. `clat remove` removes a dependency and strips its entry from the lock file.

29.6 The Package Registry and Cache

29.6.1 The Lattice Registry

Lattice's default package registry lives at:

```
https://registry.lattice-lang.org/v1
```

The package manager communicates with the registry using two HTTP endpoints:

Table 29.4: Registry API endpoints

Endpoint	Purpose
`GET /packages/⟨name⟩/versions`	List available versions (JSON)
`GET /packages/⟨name⟩/⟨version⟩`	Download package source

The versions endpoint returns JSON like `\{"versions":["1.0.0","1.1.0","2.0.0"]\}`. The download endpoint returns either a single `main.lat` file or a TOML bundle, depending on the package.

29.6.2 Overriding the Registry

You can point the package manager at a different registry using the `LATTICE_REGISTRY` environment variable:

```
# Use a private registry
export LATTICE_REGISTRY=https://packages.mycompany.com/v1
clat install

# Use a local directory as a registry (great for testing)
export LATTICE_REGISTRY=file:///home/user/my-packages
clat install
```

The `file://` protocol is particularly useful for development. If you set `LATTICE_REGISTRY=file:///path/to/packages`, the package manager copies packages directly from that directory into `lat_modules/` without any HTTP requests.

Plain HTTP Is Refused

Since v0.4.0, a network registry must be HTTPS. Point `LATTICE_REGISTRY` at an `http://` URL and the package manager ignores it, tells you why, and falls back to the default HTTPS registry:

```
$ LATTICE_REGISTRY=http://packages.example.com/v1 clat install
warning: ignoring insecure http:// LATTICE_REGISTRY (set
LATTICE_INSECURE_REGISTRY=1 to allow); using default HTTPS registry
```

The rationale is blunt: over plain HTTP, anyone between you and the registry chooses what code you install. If you genuinely need it—a registry on `localhost` during development, an air-gapped mirror—set `LATTICE_INSECURE_REGISTRY=1` and the URL is honored:

```
export LATTICE_REGISTRY=http://127.0.0.1:8080/v1
export LATTICE_INSECURE_REGISTRY=1
clat install
```

The `file://` form is unaffected by this rule—it never touches the network.

29.6.3 The Global Package Cache

When a package is downloaded from the registry, it is stored in a global cache before being copied into your project's `lat_modules/`:

```
~/.lattice/packages/<name>/<version>/
```

For example, `http-client` version `1.2.0` would be cached at:

```
~/.lattice/packages/http-client/1.2.0/
```

The next time any project on your machine needs that same package at that same version, the package manager copies it from the cache instead of downloading it again. You will see the message `(cached http-client@1.2.0)` in the install output when this happens.

Clearing the Cache

If you ever need to force a fresh download, delete the cached package directory:

```
rm -rf ~/.lattice/packages/http-client/1.2.0
clat install
```

Or clear the entire cache:

```
rm -rf ~/.lattice/packages
```

29.6.4 The `lat_modules/` Directory

After installation, your project's dependency tree lives in `lat_modules/`:

```
my-project/
  lattice.toml
  lattice.lock
  main.lat
  lat_modules/
    http-client/
      lattice.toml
      main.lat
    json-schema/
      lattice.toml
      src/
        main.lat
    cli-args/
      lattice.toml
      main.lat
```

Each package directory contains at minimum a `lattice.toml` and one or more `.lat` source files. The module resolution system (described in Section 28.5) knows how to find the entry point for each package.

To use an installed package in your code, import it by its package name:

Listing 29.8: Importing installed packages

```
import "http-client" as http
import { validate } from "json-schema"
import "cli-args" as args

let response = http.get("https://api.weather.gov/points/44.9,-93.2")
let data = validate(response.body, schema)
```

The import system tries the built-in modules first, then falls back to `lat_modules/`—exactly as described in Section 28.5.

29.7 Dependency Graphs and Cycle Detection

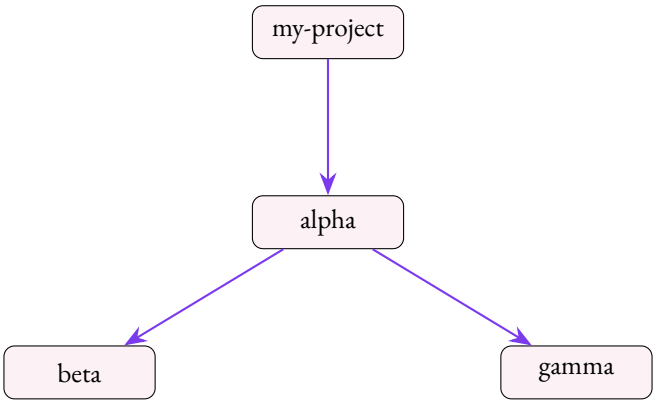
When packages depend on other packages, those dependencies form a directed graph. Lattice builds this graph after installation and checks it for cycles.

29.7.1 How the Graph Is Built

The dependency graph is constructed by reading each installed package's `lattice.toml`:

1. Your project is the root node.
2. Each of your direct dependencies becomes a node with an edge from the root.
3. For each dependency, the package manager reads *its* `lattice.toml` and adds edges for its dependencies.
4. This continues via breadth-first search until all transitive dependencies have been discovered.

For example, if your project depends on `alpha`, and `alpha` depends on `beta` and `gamma`, the graph looks like:



29.7.2 Cycle Detection

Circular dependencies—where `A` depends on `B`, `B` depends on `C`, and `C` depends on `A`—are a serious problem. They make it impossible to determine a safe installation and loading order.

Lattice detects cycles using a depth-first search (DFS) algorithm. Each node is colored during traversal:

- **White**: not yet visited.
- **Gray**: currently being explored (on the DFS stack).
- **Black**: fully explored (all descendants processed).

If the DFS encounters a gray node—meaning we have followed a path back to a node that is still being explored—a cycle has been found. The algorithm then traces back through the DFS stack to produce a human-readable cycle path:

```
error: circular dependency detected: alpha -> beta -> gamma -> alpha
```

You can see this implementation in `src/package.c`, in the `dfs_visit()` and `pkg_dep_graph_has_cycle()` functions.

Circular Dependencies Are Fatal

If a circular dependency is detected, `clat install` fails and does not write a lock file. You must resolve the cycle—typically by refactoring one of the packages to remove the circular relationship—before installation can succeed.

29.8 Trusting Your Dependencies

A package manager is a program that downloads code from strangers and puts it where your interpreter will run it. That is a lot of trust to hand over, and v0.4.0 spent most of its package-manager budget narrowing it. Three of the four hardening measures are invisible when everything is well-behaved and loudly visible when something is not.

We have already met the first two: HTTPS-only registries (Section 29.6.2) and lock-file checksums (Section 29.5.2). Here are the other two.

29.8.1 Manifest Fields Are Validated Before Use

A dependency name becomes a directory name: `lat_modules/⟨name⟩`. A version string becomes a cache path component: `~/.lattice/packages/⟨name⟩/⟨version⟩`. A git tag becomes part of a cache path too. Every one of those is an opportunity to write a file somewhere it does not belong—if nobody checks.

Lattice checks. Dependency names must be non-empty and drawn from `[A-Za-z0-9._-]`; versions and git refs must be safe single path components (no separators, no control bytes, no embedded `..`, and never `.` or `..` on their own). Anything else is rejected before a single byte of I/O happens:

Listing 29.9: Four manifests, four refusals

```
# [dependencies] "../../../etc/cron.d/pwn" = "1.0.0"
error: invalid or unsafe dependency entry '../../../etc/cron.d/pwn' in lattice.toml

# [dependencies] harmless = "../../../etc"
error: invalid or unsafe dependency entry 'harmless' in lattice.toml

# [dependencies] harmless = { git = "...", tag = "../../../../etc" }
error: invalid or unsafe dependency entry 'harmless' in lattice.toml

# [dependencies] "pkg; rm -rf ~" = "1.0.0"
error: invalid or unsafe dependency entry 'pkg; rm -rf ~' in lattice.toml
```

The same suspicion is applied to values the *registry* sends back. A registry that answers a version query with `"../../etc/foo"` gets told off, and the install fails rather than falling back to something "close enough":

Listing 29.10: A hostile registry response, refused

```
$ clat install
  Installing colorfmt@*...  error: package 'colorfmt' not found locally;
  registry fetch also failed: registry returned unsafe version string
  '../../etc/foo' for package 'colorfmt'

1 package(s) failed to install.
 FAILED
```

There is one more confinement worth knowing about, because it can bite you honestly rather than maliciously. A package's `entry` field is resolved and then checked against the package directory: it must land *inside* `lat_modules/⟨name⟩/`.

Listing 29.11: An entry that escapes the package directory is not loaded

```
# lat_modules/sneaky/lattice.toml
#   entry = "../../secret.lat"
$ clat main.lat
vm error: import: cannot find 'sneaky.lat'
```

A legitimate sub-path such as `entry = "src/main.lat"` resolves inside the package directory and imports normally.

29.8.2 No Shell, Ever

The package manager runs external programs—`git` to clone, `cp` to copy trees, `rm` to clean up. Historically that is exactly where package managers get owned: a string from a manifest gets interpolated into a shell command line, and a semicolon in a dependency name becomes arbitrary code execution.

Since v0.4.0, no part of the package manager invokes a shell. Every external command goes through `fork()` plus `execvp()` with an explicit argument vector, and each argument vector includes a literal `--` separator before any manifest-derived value:

```
cp  -R  --  <src>  <dst>
rm  -rf --  <path>
git clone ... -- <url> <dir>
```

The `--` stops a value that begins with a dash from being read as an option, and the absence of a shell means metacharacters are inert. A dependency literally named `"pkg; rm -rf ~"` would be a filename, not a command—and as we saw above, the name validator rejects it long before it gets that far. You can read the whole surface in `src/package.c`; it is short on purpose.

These Checks Are Not a Sandbox

Everything above stops a malicious *manifest* from escaping the directories the package manager owns. None of it stops a malicious *package* from doing whatever it likes once you `import` it—installed code runs with your full privileges, exactly like code you wrote. Read what you depend on, and pin it with a tag or a revision so that "what you read" and "what you run" stay the same thing.

29.9 Under the Hood: The Install Pipeline

For readers who want the full picture, here is what happens when you run `clat install`, step by step.

1. **Read `lattice.toml`.** Parse the TOML into a `PkgManifest` structure containing the project metadata and dependency list. If the file does not exist, print an error suggesting `clat init`.
2. **Check for an existing lock file.** If `lattice.lock` exists, parse it into a `PkgLock` structure. The locked versions will be preferred during resolution.
3. **Create `lat_modules/`.** If the directory does not exist, create it.
4. **Resolve each dependency.** For each entry in `[dependencies]`:
   - If the lock file has an entry for this package, and the locked version satisfies the manifest constraint, use the locked version.
   - Check if `lat_modules/⟨name⟩` already exists. If so, verify its version.
   - If not installed: try a `file://` registry (if configured), then fall back to the HTTP registry.
   - On the HTTP registry: query available versions, select the best match, check the global cache (`~/.lattice/packages/`), download if needed, copy to `lat_modules/`.
5. **Verify integrity.** Hash the installed package tree with SHA-256 and compare it against the checksum recorded in `lattice.lock`. On a mismatch, mark the package `FAILED` and report the integrity error (Section 29.5.2).
6. **Build the dependency graph.** Read each installed package's `lattice.toml` to discover transitive dependencies. Construct a directed graph using BFS.
7. **Check for cycles.** Run DFS-based cycle detection on the graph. If a cycle is found, report it and abort.

8. **Write `lattice.lock`.** Serialize the resolved versions, sources, and checksums to TOML.

Flat Installation

Lattice uses a *flat* dependency layout: all packages live directly under `lat_modules/`, not nested inside each other. This means that if two packages depend on the same third package, there is only one copy. The trade-off is that version conflicts between transitive dependencies are not yet automatically resolved—this is a known limitation that will be addressed in a future version of the package manager.

29.10 Putting It All Together

Let's walk through a complete workflow from project creation to running with dependencies.

Listing 29.12: Complete project workflow

```
# Create a new project
mkdir sensor-dashboard && cd sensor-dashboard
clat init

# Add dependencies
clat add json-utils ^1.0.0
clat add http-server ^0.5.0
clat add template-engine ~2.1.0
```

Your `lattice.toml` now looks like:

```
[package]
name = "sensor-dashboard"
version = "0.1.0"

[dependencies]
json-utils = "^1.0.0"
http-server = "^0.5.0"
template-engine = "~2.1.0"
```

Write your application:

Listing 29.13: main.lat — using installed packages

```
import "json-utils" as json
import "http-server" as server
import { render } from "template-engine"

import "sensors/temperature" as temp
import "sensors/humidity" as humid

fn handle_dashboard(req: any) -> any {
    let readings = [
        temp.read_sensor(),
        humid.read_sensor()
    ]
    let body = render("dashboard.html", json.stringify(readings))
    return server.response(200, body)
}

let app = server.create(8080)
server.route(app, "/", handle_dashboard)
server.start(app)
print("Dashboard running on http://localhost:8080")
```

Notice how third-party packages (from `lat_modules/`) and your own modules (from `sensors/`) are imported with the same syntax. The module system handles the resolution transparently.

Run the program:

```
clat main.lat
```

Share the project with a colleague:

```
# In .gitignore:
lat_modules/

# Commit everything else:
git add lattice.toml lattice.lock main.lat sensors/
git commit -m "Initial sensor dashboard"
git push
```

When your colleague clones the repository, they run:

```
clat install
clat main.lat
```

The lock file ensures they get the exact same dependency versions you used.

29.11 Exercises

1. **Initialize and explore.** Run `clat init` in a new directory. Examine the generated `lattice.toml`. Try running `clat init` again—what happens? Now manually add a `description` and `license` field to the manifest.

2. **Semver reasoning.** Given the available versions `[1.0.0, 1.1.0, 1.2.3, 1.3.0, 2.0.0, 2.1.0]`, which version does each constraint select?

   - `"*"`
   - `"^1.1.0"`
   - `"~1.2.0"`
   - `">=1.3.0"`
   - `"<=1.2.3"`
   - `"1.2.3"`

3. **Local file registry.** Create a directory called `my-registry` containing a subdirectory `greetings` with a `main.lat` that exports a `hello` function. Set `LATTICE_REGISTRY=file:///path/to/my-registry` and run `clat add greetings`. Verify that the package appears in `lat_modules/` and can be imported.

4. **Lock file investigation.** Add a dependency with a caret constraint. Examine the generated `lattice.lock`. Change the version constraint in `lattice.toml` to a tilde constraint for the same minor version. Run `clat install` again. Does the lock file change? Why or why not?

5. **Circular dependency detection.** Create three packages (`alpha`, `beta`, `gamma`) in a local file registry. Make `alpha` depend on `beta`, `beta` depend on `gamma`, and `gamma` depend on `alpha`. Create a project that depends on `alpha` and run `clat install`. Observe the error message. Then fix the cycle by removing one of the dependency edges and verify that the install succeeds.

6. **Trip the integrity check.** Install a package from a local file registry and note the `checksum` recorded in `lattice.lock`. Append a comment line to one of the installed files under `lat_modules/` and run `clat install` again. Confirm that the install fails and that your edit

is still there afterward. Now remove that package's entry from the lock file and install once more—what does the new checksum tell you?

7. **Try to escape.** In a scratch directory, write a `lattice.toml` whose dependency name is `"../../../tmp/escape"`, and another whose version string contains a `/`. Run `clat install` on each and record the exact error. Then set `LATTICE_REGISTRY` to an `http://` URL and observe what the package manager does with it.

8. **Pin a git dependency.** Create a local git repository containing a one-function Lattice package, tag it `v0.1.0`, and add it to a project with `clat add ... --git ... --tag v0.1.0`. Inspect the resulting manifest entry, the lock entry, and the contents of `lat_modules/`—is there a `.git` directory? Now commit a change upstream, re-tag, and reason about what a `--branch` dependency would have done instead.

What's Next

With modules, imports, and packages under your belt, you have everything you need to build and share Lattice projects of any size. But writing code is only half the story—maintaining it is the other half. In Chapter 30, we will explore Lattice's built-in code formatter and documentation generator, tools that help keep your codebase consistent and well-documented as it grows.

Part IX

Tooling and Developer Experience

Chapter 30

The Formatter and Doc Generator

Every codebase drifts. One developer uses tabs, another uses spaces. Someone wraps function arguments at 80 columns; someone else at 120. Doc comments get written in three different styles. A week later, the "git blame" for every file reads like a United Nations assembly. Lattice solves this with two built-in tools: `clat fmt` for formatting source code and `clat doc` for generating documentation. Both are first-class citizens of the `clat` CLI—no third-party installation required.

30.1 `clat fmt` — Formatting Your Code

Let's start with a messy file. Suppose a colleague has handed us `sensor.lat` with inconsistent spacing, missing indentation, and some creative use of whitespace:

Listing 30.1: Before formatting: messy whitespace and indentation

```
fn read_temperature(  sensor_id:Int,unit: String   ) -> Float{
    let raw=sensor_id *   0.48+2.1
if unit ==  "celsius"{
return raw
}else{
    return raw*9.0/5.0+32.0
}}
```

One command brings order to the chaos:

Listing 30.2: Running clat fmt

```
$ clat fmt sensor.lat
```

It prints nothing and exits 0. Silence is the formatter's way of saying "done"—the file has been rewritten in place:

Listing 30.3: After formatting: clean and consistent

```
fn read_temperature(sensor_id: Int, unit: String) -> Float {
    let raw = sensor_id * 0.48 + 2.1
    if unit == "celsius" {
        return raw
    } else {
        return raw * 9.0 / 5.0 + 32.0
    }
}
```

Notice what the formatter did:

- Normalized spacing around operators (=, *, +) to exactly one space on each side.
- Enforced the *attached brace style*: the opening `\{` appears on the same line as the `fn`, `if`, and `else` keywords, preceded by a single space.
- Inserted a space after the colon in type annotations (`sensor_id: Int`) and after commas.
- Applied consistent four-space indentation at each nesting level.
- Placed the `else` clause on the same line as the closing brace of the `if` block (`} else {`).
- Ensured a single trailing newline at the end of the file.

clat fmt

The `clat fmt` command reads a Lattice source file, reformats it using the canonical style, and writes the result back. It operates on a character-by-character scan of the source (implemented in `src/formatter.c`), preserving all comments and string contents while normalizing whitespace and indentation. The indentation width is four spaces.

Here is the whole command surface, straight from `clat fmt --help`:

Listing 30.4: clat fmt –help

```
Usage: clat fmt [options] <file.lat>

Format Lattice source code.

Options:
  -w, --width <N>   Set max line width (default: 100)
  --check           Check formatting without modifying
  --stdin           Read from stdin, write to stdout
  -h, --help        Show this help
```

30.1.1 One File at a Time

Note the singular `<file.lat>` in that usage line: the formatter takes exactly one path. Hand it two and it declines:

```
$ clat fmt src/main.lat src/utils.lat
usage: clat fmt [--check] [--stdin] [--width N] [file.lat]
```

Directories are not walked either. To format a whole tree, let your shell do the walking:

Listing 30.5: Formatting a project tree

```
find src lib -name '*.lat' -exec clat fmt {} \;
```

30.1.2 Check Mode: CI-Friendly Verification

In a continuous integration pipeline, you don't want to *rewrite* files—you want to *verify* that they're already formatted. The `--check` flag does exactly this:

Listing 30.6: Using --check

```
$ clat fmt --check sensor.lat
sensor.lat: not formatted
$ echo $?
1
```

A file that is already canonical prints nothing and exits 0. `--check` reports *whether* a file differs, not *how*—there is no diff. When you want to see the difference, format a copy and diff it yourself.

Under the hood, `--check` calls the `lat_format_check()` function (see `include/formatter.h`), which formats the source and compares the result to the original. If they match, the file is already canonical.

Add Formatting to Your CI Pipeline

Because `--check` handles one file and exits non-zero on the first unformatted one, the usual CI incantation is a loop:

```
find src lib -name '*.lat' -print0 |
  xargs -0 -n1 clat fmt --check
```

This ensures every pull request follows the same style. No more bikeshedding in code reviews about where the braces go.

30.1.3 Reading from Standard Input

The formatter can also read from standard input with the `--stdin` flag. This is useful for editor integrations and piped workflows:

Listing 30.7: Formatting via stdin

```
$ clat fmt --stdin < messy.lat
fn read_temperature(sensor_id: Int, unit: String) -> Float {
    let raw = sensor_id * 0.48 + 2.1
    ...
```

The `lat_format_stdin()` function in `src/formatter.c` reads all of standard input into a buffer, formats it, and writes the result to standard output. This is also the form to reach for when you want to *inspect* what the formatter would do without touching the file on disk.

30.1.4 Doc Comments Survive, and Formatting Is Idempotent

Two properties matter more than any individual spacing rule, and both have been guaranteed since v0.4.1.

The first: `///` doc comments (Section 30.3) are preserved exactly, including the blank `///` lines that separate paragraphs. A formatter that silently reflowed your documentation would be worse than no formatter at all.

The second: formatting is *idempotent*. Running `clat fmt` on already-formatted source produces byte-identical output.

Listing 30.8: Verifying idempotence for yourself

```
$ clat fmt --stdin < messy.lat  > pass1.lat
$ clat fmt --stdin < pass1.lat  > pass2.lat
$ diff pass1.lat pass2.lat && echo identical
identical
```

Idempotence is what makes --check meaningful: if a second pass could keep changing the file, "already formatted" would not be a stable state, and your CI job would flip-flop.

30.2 What the Formatter Does Not Do: `--width`

Every formatter chapter eventually reaches the part about line wrapping. This one reaches it early, because Lattice's answer is unusually short.

The `-w` / `--width` flag exists, it is documented in `clat fmt --help`, and it defaults to 100 (the `DEFAULT_LINE_WIDTH` constant in `src/formatter.c`). What it does *not* do, as of v0.6.4, is change the output.

Let's prove that to ourselves with a signature nobody would call short:

Listing 30.9: A function with many parameters

```
fn create_report(title: String, author: String, created_at: String, tags: Array, body:
    String, draft: Bool) -> Map {
    let report = Map::new()
    report["title"] = title
    return report
}
```

That signature is 116 columns wide. Run it through the formatter at 60, 80, 100, and 120 columns and all four passes return it untouched, on one line:

Listing 30.10: Four widths, one result

```
$ for w in 60 80 100 120; do clat fmt --width $w --stdin < wide.lat; done
# ... the same 113-column signature, four times ...
```

The Formatter Is a Normalizer, Not a Re-Wrapper

Lattice's formatter is a single character-by-character pass that fixes spacing, indentation, and brace placement. It never moves a construct from one line to another, which is precisely why it can guarantee idempotence and doc-comment preservation so cheaply. The width value is carried into the formatter's state and reserved for a future wrapping pass; today no code reads it. Passing `--width` is harmless, and it is honest to leave it out.

The practical consequence: line length is on you.

Hand-Split Signatures Come Back Flush Left

The formatter keeps your line *breaks*, but it computes indentation from brace nesting alone—and a parameter list opened with `(` does not count as a nesting level. Split a signature by hand and the continuation lines are pushed to column zero:

Listing 30.11: Before: hand-indented parameters

```
fn create_report(
    title: String,
    author: String,
    tags: Array
) -> Map {
    return Map::new()
}
```

Listing 30.12: After clat fmt: the indentation is gone

```
fn create_report(
title: String,
author: String,
tags: Array
) -> Map {
    return Map::new()
}
```

Until the formatter grows a wrapping pass, the friendliest option for a long signature is to leave it on one line and let your editor soft-wrap it. If a multi-line signature matters more to you than formatter-clean output, keep that file out of your `-check` loop.

30.3 /// Doc Comments

Lattice uses triple-slash (`///`) comments for documentation. Unlike regular `//` comments, doc comments are extracted by the documentation generator and attached to the declaration that follows them.

Listing 30.13: Doc comments on a function

```
/// Calculate the body mass index for a patient.
/// Returns the BMI as a Float value.
fn calculate_bmi(weight_kg: Float, height_m: Float) -> Float {
    return weight_kg / (height_m * height_m)
}
```

Doc comments can span multiple consecutive lines. The `///` prefix and one optional space after it are stripped, and the remaining text becomes the documentation body.

30.3.1 Documenting All Declaration Types

Doc comments work on every kind of top-level declaration: functions, structs, enums, traits, impl blocks, and even variables.

Listing 30.14: Doc comments on structs and enums

```
/// A patient record in the healthcare system.
/// Contains demographic and medical information.
struct Patient {
    /// The patient's full legal name.
    name: String,
    /// Age in years.
    age: Int,
    /// Blood type (e.g., "A+", "O-").
    blood_type: String
}

/// Possible outcomes of a medical test.
enum TestResult {
    /// The test came back normal.
    Negative,
    /// The test detected the condition.
    Positive,
    /// The sample was insufficient or contaminated.
    Inconclusive
}
```

Notice that each struct field and enum variant can have its own doc comment. The documentation generator (covered in the next section) extracts these and associates them with the correct field or variant.

30.3.2 Module-Level Doc Comments

If a `///` comment block appears at the very top of a file—before any declaration—it is treated as a *module-level* doc comment. This is the place to describe what the file or module does:

Listing 30.15: Module-level documentation

```
/// The geometry module provides functions for calculating
/// areas and perimeters of common shapes.
///
/// All functions accept dimensions in the same unit
/// and return results in squared units where applicable.

fn circle_area(radius: Float) -> Float {
    return 3.14159265 * radius * radius
}

fn rectangle_area(width: Float, height: Float) -> Float {
    return width * height
}
```

The doc extractor in `src/doc_gen.c` distinguishes module-level docs from declaration docs by checking whether the doc block is followed by a declaration keyword (`fn`, `struct`, `enum`, `trait`, `impl`, `flux`, `fix`, `let`, or `export`). If not, it's treated as the module description.

30.3.3 Doc Comment Conventions

While Lattice doesn't enforce a specific doc comment format, the community has settled on a few conventions:

Listing 30.16: Recommended doc comment style

```
/// Open a database connection to the given host.
///
/// The connection uses a pooled TCP socket under the hood.
/// If the host is unreachable, returns an error after
/// the specified timeout.
///
/// Parameters are validated before connecting. An empty
/// host string will produce an immediate error.
fn connect(host: String, port: Int, timeout_ms: Int) -> Map {
    // ... implementation
    return Map::new()
}
```

- First line: a brief one-sentence summary.

- Blank line (an empty `///` line) separating the summary from the longer description.

- Additional paragraphs for context, usage notes, or caveats.

Regular Comments Are Ignored

Only `///` comments are picked up by `clat doc`. Regular `//` comments and block comments (`/* ... */`) are treated as implementation notes and excluded from generated documentation. If you write documentation with `//` instead of `///`, it won't appear in the output.

30.4 `clat doc` — Generating Documentation

The `clat doc` command extracts doc comments from one or more `.lat` files and renders them as human-readable documentation. Unlike `clat fmt`, it happily takes several paths—files, directories, or a mix:

Listing 30.17: clat doc –help

```
Usage: clat doc [options] <file.lat|dir>...

Generate documentation from Lattice source files.

Options:
  --md, --markdown          Markdown output (default)
  --text                    Plain text output
  --json                    JSON output
  --html                    HTML output
  --doc-format FORMAT       Set format (text, md, json, html)
  -o, --output DIR          Write output files to DIR
  -h, --help                Show this help
```

By default, this prints Markdown to standard output. Let's walk through a complete example. Suppose we have `math_utils.lat`:

Listing 30.18: A well-documented utility module

```
/// Utility functions for common mathematical operations.

/// Clamp a value to the range [low, high].
/// If the value is below low, returns low.
/// If above high, returns high.
fn clamp(value: Float, low: Float, high: Float) -> Float {
    if value < low {
        return low
    }
    if value > high {
        return high
    }
    return value
}

/// Linear interpolation between two values.
/// When t is 0.0, returns a. When t is 1.0, returns b.
fn lerp(a: Float, b: Float, t: Float) -> Float {
    return a + (b - a) * t
}

/// A 2D point in Cartesian space.
struct Point {
    /// Horizontal coordinate.
    x: Float,
    /// Vertical coordinate.
    y: Float
}

/// Compute the Euclidean distance between two points.
fn distance(p1: Point, p2: Point) -> Float {
    let dx = p2.x - p1.x
    let dy = p2.y - p1.y
    return (dx * dx + dy * dy) ** 0.5
}
```

Running `clat doc math_utils.lat` produces exactly this:

Listing 30.19: Generated Markdown output

```
Utility functions for common mathematical operations.

## Functions

### `clamp(value: Float, low: Float, high: Float) -> Float`

Clamp a value to the range [low, high].
If the value is below low, returns low.
If above high, returns high.

### `lerp(a: Float, b: Float, t: Float) -> Float`

Linear interpolation between two values.
When t is 0.0, returns a. When t is 1.0, returns b.

### `distance(p1: Point, p2: Point) -> Float`

Compute the Euclidean distance between two points.

## Structs

### `struct Point`

A 2D point in Cartesian space.

| Field | Type | Description |
|-------|------|-------------|
| `x` | `Float` | Horizontal coordinate. |
| `y` | `Float` | Vertical coordinate. |
```

Two things to notice. The module-level doc comment became the opening paragraph, with no heading of its own. And the generator *groups items by kind*—all three functions appear together under `## Functions`, even though `distance` was written below the `Point` struct in the source. The kinds are, in order: functions, structs, enums, traits, implementations, and variables.

30.4.1 Documenting a Directory

Pass a directory to document all `.lat` files inside it:

Listing 30.20: Documenting an entire directory

```
$ clat doc src/
# geometry.lat

Geometry helpers.

## Functions

### `circle_area(radius: Float) -> Float`

Area of a circle.

---

# math_utils.lat

Utility functions for common mathematical operations.

...
```

When more than one file is processed, each gets an # heading naming the file, and files are separated by a horizontal rule. Files are visited in directory order, so the output is stable across runs.

30.4.2 Writing Output to Files

For larger projects, you'll want to write documentation to files rather than standard output. The `-o` (or `--output`) flag specifies an output directory:

Listing 30.21: Writing docs to a directory

```
$ clat doc -o docs/ src/
wrote /home/you/project/docs/geometry.md
wrote /home/you/project/docs/math_utils.md
```

Each input file produces a corresponding output file, and `clat` reports the absolute path it wrote. The `.lat` extension is replaced with the appropriate extension for the chosen format (`.md` for Markdown, `.json` for JSON, `.html` for HTML, or `.txt` for plain text). The output directory is created automatically if it doesn't exist.

30.5 Output Formats: Markdown, JSON, HTML

The documentation generator supports four output formats, each suited to a different use case.

30.5.1 Markdown (Default)

Markdown is the default because it's the lingua franca of technical documentation—readable as plain text, renderable on GitHub, GitLab, and most wikis.

Listing 30.22: Generating Markdown output

```
// $ clat doc --md src/server.lat
// (or simply: clat doc src/server.lat)
```

The Markdown renderer produces headings, code-formatted signatures, tables for struct fields, and bulleted lists for enum variants.

30.5.2 Plain Text

For terminal viewing or piping to other tools:

Listing 30.23: Generating plain text output

```
// $ clat doc --text src/server.lat
```

The text format uses simple labels like `FUNCTIONS`, `STRUCTS`, and `ENUMS`, with dashed underlines and indented descriptions.

30.5.3 JSON

The JSON format is designed for machine consumption. It's what you'd use to build a custom documentation website, IDE plugin, or search index:

Listing 30.24: Generating JSON output

```
// $ clat doc --json src/math_utils.lat
```

The output is a JSON array where each element represents a source file. Each file object contains `"file"`, an optional `"module_doc"`, and an `"items"` array. Each item has a `"kind"` (`"function"`, `"struct"`,

`"enum"`, `"trait"`, `"impl"`, or `"variable"`), a `"name"`, a `"line"` number, and optional `"doc"` text, plus kind-specific fields like `"params"` for functions or `"fields"` for structs.

Listing 30.25: Example JSON output (abbreviated)

```
// [
//   {
//     "file": "math_utils.lat",
//     "module_doc": "Utility functions for ...",
//     "items": [
//       {
//         "kind": "function",
//         "name": "clamp",
//         "line": 4,
//         "doc": "Clamp a value to the range ...",
//         "params": [
//           {"name": "value", "type": "Float"},
//           {"name": "low", "type": "Float"},
//           {"name": "high", "type": "Float"}
//         ],
//         "return_type": "Float"
//       }
//     ]
//   }
// ]
```

Building a Documentation Website

The JSON output pairs well with static site generators. Extract docs with `clat doc --json src/ > api.json`, then feed `api.json` into your favorite template engine to produce a searchable HTML site.

30.5.4 HTML

For a self-contained, styled documentation page:

Listing 30.26: Generating HTML documentation

```
// $ clat doc --html -o docs/ src/
```

The HTML renderer produces a complete web page with a dark theme, syntax-highlighted signatures, and interactive hover effects. Function signatures are color-coded—keywords in purple, types in gold, function names in blue—making it easy to scan a large API surface.

Struct fields render as tables, enum variants as styled lists, and trait methods as indented signatures under their parent trait.

30.5.5 The `-doc-format` Flag

As an alternative to the shorthand flags, you can use `--doc-format` with an explicit format name:

Listing 30.27: Using --doc-format

```
// $ clat doc --doc-format html src/
// $ clat doc --doc-format json src/
// $ clat doc --doc-format md src/
// $ clat doc --doc-format text src/
```

This is useful in scripts where you want to parameterize the format.

30.5.6 How Extraction Works Under the Hood

The documentation extractor in `src/doc_gen.c` uses its own lightweight scanner (`DocScanner`) rather than the full Lattice lexer. This is a deliberate design choice: the scanner needs to preserve doc comments while skipping implementation details, so it only recognizes enough syntax to identify declarations.

The process works as follows:

1. The scanner reads through the source file, skipping whitespace and regular comments.
2. When it encounters a `///` line, it reads consecutive doc comment lines into a single block, stripping the prefix and one optional leading space.
3. After the doc block, the scanner looks for a declaration keyword (`fn`, `struct`, `enum`, `trait`, `impl`, or a phase keyword).
4. For each declaration, the scanner extracts the name and kind-specific details: function parameters and return types, struct fields with their types, enum variants with optional tuple parameters, trait method signatures, and impl block methods.
5. The result is a `DocFile` structure (defined in `include/doc_gen.h`) containing a list of `DocItem` values that the renderers consume.

Items that lack doc comments are still extracted for functions, structs, enums, and traits. Variables, however, are only included if they have a doc comment, since most local or temporary variables aren't part of a module's public API.

Test Blocks Are Skipped

The doc generator recognizes `test "name" \{ ... \}` blocks and skips them entirely. Tests are implementation details, not part of your public documentation.

Exercises

1. Take one of the Lattice files you've written in earlier chapters and run `clat fmt` on it. Compare the before and after. Were there any formatting choices that surprised you? Try running with `--width 80` and `--width 120` to see how the output changes.
2. Create a small library module (perhaps a `temperature.lat` with conversion functions) and document every function, struct, and constant with `///` comments. Then run `clat doc --html -o docs/ temperature.lat` and open the resulting HTML in a browser.
3. Write a shell script (or Makefile target) that runs `clat fmt --check` on all `.lat` files in a directory tree and exits with a non-zero code if any file needs formatting. This is the skeleton of a CI formatting check.
4. Generate JSON documentation for a multi-file project and write a small Lattice script that reads the JSON and counts the total number of documented functions, structs, and enums across all files.

What's Next

With clean formatting and thorough documentation in hand, the natural next step is to make sure everything actually *works*. In Chapter 31, we'll explore Lattice's built-in testing system—inline `test` blocks that live right alongside your code, a rich set of assertion functions, and the `clat test` command that discovers and runs your entire test suite.

Chapter 31

Testing

Untested code is code that's waiting to betray you. You might trust it today—it ran once, it printed the right number—but refactor a function, add a parameter, swap a data structure, and suddenly the thing that "worked" doesn't. Lattice takes testing seriously: the language has built-in syntax for tests, a CLI runner that discovers and executes them, and a full suite of assertion functions that produce clear failure messages. No external framework to install, no configuration file to wrangle. Tests live right next to the code they verify.

31.1 `test "name" { body }` — Inline Tests

Lattice's `test` keyword lets you write tests directly in your source files. The syntax is:

Listing 31.1: Basic test block syntax

```
test "addition works" {
    assert(2 + 2 == 4, "basic arithmetic")
}
```

Test Block

A *test block* is a top-level declaration of the form `test "name" \{ body \}`. The name is a string literal that describes what the test verifies. The body contains any number of statements—typically assertions—that run when the test suite is invoked via `clat test`. Test blocks are *completely ignored* during normal program execution (`clat program.lat`).

There Is No `clat run`

Lattice runs a file by naming it directly: `clat program.lat`. There is no `run` subcommand—`clat run program.lat` reports `error: cannot read 'run'` and exits 1, because `run` is interpreted as the filename. The subcommands are `compile`, `test`, `fmt`, `doc`, `init`, `install`, `add`, and `remove`; everything else is a path.

Let's write a more realistic example. Suppose we have a temperature conversion module:

Listing 31.2: Temperature conversion with inline tests

```
fn celsius_to_fahrenheit(c: Float) -> Float {
    return c * 9.0 / 5.0 + 32.0
}

fn fahrenheit_to_celsius(f: Float) -> Float {
    return (f - 32.0) * 5.0 / 9.0
}

test "freezing point conversion" {
    assert_eq(celsius_to_fahrenheit(0.0), 32.0)
    assert_eq(fahrenheit_to_celsius(32.0), 0.0)
}

test "boiling point conversion" {
    assert_eq(celsius_to_fahrenheit(100.0), 212.0)
    assert_eq(fahrenheit_to_celsius(212.0), 100.0)
}

test "body temperature" {
    let body_temp_f = celsius_to_fahrenheit(37.0)
    // 37.0 C should be approximately 98.6 F
    assert(body_temp_f > 98.5, "too low")
    assert(body_temp_f < 98.7, "too high")
}
```

The tests sit right below the functions they exercise. Anyone reading this file immediately understands what the functions are expected to do.

Running that file gives us exactly what we hoped for:

Listing 31.3: The temperature suite passing

```
$ clat test temperature.lat
  PASS  freezing point conversion
  PASS  boiling point conversion
  PASS  body temperature

Results: 3 passed, 0 failed (3 total)
```

31.1.1 Tests Are Top-Level Declarations

Test blocks are parsed as top-level items by the Lattice parser (handled in `src/parser.c` as `ITEM_TEST`). They cannot appear inside functions, loops, or other blocks. This keeps them at the module level, visible and discoverable.

Tuck one inside a function and the parser refuses:

Listing 31.4: A test block inside a function is a parse error

```
$ clat nested.lat
error: 2:5: parse error: unexpected token 'test' in expression
4:1: parse error: unexpected token '}' in expression
```

Two errors from one mistake—the parser recovers and keeps going rather than stopping at the first complaint, so a single run tells you about everything it found.

Listing 31.5: Tests belong at the top level

```
fn compute(x: Int) -> Int {
    return x * x
}

// This is correct --- test at the top level:
test "compute squares" {
    assert_eq(compute(5), 25)
    assert_eq(compute(0), 0)
    assert_eq(compute(-3), 9)
}
```

31.1.2 Tests Can Use Everything in the File

When `clat test` runs, it first executes all non-test top-level code (imports, function definitions, struct declarations, variable bindings) before running any test block. This means tests have access to every function, struct, enum, and global defined in the file.

Listing 31.6: Tests have access to all module-level definitions

```
struct Color {
    r: Int,
    g: Int,
    b: Int
}

fn mix(c1: Color, c2: Color) -> Color {
    return Color {
        r: (c1.r + c2.r) / 2,
        g: (c1.g + c2.g) / 2,
        b: (c1.b + c2.b) / 2
    }
}

test "mixing colors" {
    let red = Color { r: 255, g: 0, b: 0 }
    let blue = Color { r: 0, g: 0, b: 255 }
    let purple = mix(red, blue)

    assert_eq(purple.r, 127)
    assert_eq(purple.g, 0)
    assert_eq(purple.b, 127)
}
```

Tests Run in Their Own Scope

Each test block runs in a fresh scope—`evaluator_run_tests()` brackets every block with `env_push_scope()` and `env_pop_scope()` (`src/eval.c`). Variables defined inside one test are not visible to other tests. This prevents accidental coupling between test cases—each test is independent.

31.1.3 What Happens During Normal Execution?

When you run a file directly, test blocks are skipped entirely. The compiler (in `src/stackcompiler.c`) treats `ITEM_TEST` items as no-ops during normal compilation, and the same holds under `--tree-walk` and `--regvm`. This means you can sprinkle tests throughout your source code without any runtime cost in production.

Listing 31.7: Tests are invisible during normal execution

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

test "greeting includes the name" {
    assert_contains(greet("Lattice"), "Lattice")
}

print(greet("World"))
```

Listing 31.8: The same file, run and tested

```
$ clat greeter.lat
Hello, World!

$ clat test greeter.lat
Hello, World!
  PASS  greeting includes the name

Results: 1 passed, 0 failed (1 total)
```

Look closely at the second invocation: `Hello, World!` appears there too. `clat test` executes every non-test top-level statement before it runs a single test, so top-level `print` calls fire during the test run as well. If you want a file that stays quiet under test, keep its side effects inside functions.

31.2 `clat test` — Running the Test Suite

The `clat test` command discovers and runs tests. Point it at one file and it prints a line per test and a tally:

Listing 31.9: Running tests in a single file

```
$ clat test temperature.lat
  PASS  freezing point conversion
  PASS  boiling point conversion
  PASS  body temperature

Results: 3 passed, 0 failed (3 total)
```

The Test Runner Is the Tree-Walking Interpreter

`clat test` does not go through the bytecode VM. `run_test_file_ex()` in `src/main.c` builds a fresh `Evaluator` and hands the program to `evaluator_run_tests()` in `src/eval.c`. That is why `clat test` refuses `--tree-walk` and `--regvm`—there is no backend to pick. Any flag outside the table in Section 31.2.4 is rejected with the usage line.

The practical consequence: a test suite that passes tells you the *tree-walker* agrees with you. When a chapter of this book flags a backend divergence, exercise the other backends by running a plain `.lat` file under `--regvm` or the default VM, not through `clat test`.

31.2.1 Directory Discovery

When you pass a directory instead of a file, `clat test` recursively searches for test files. The naming convention (encoded in `is_test_file()` in `src/main.c`) is:

- Files matching `test_*.lat` (e.g., `test_math.lat`, `test_parser.lat`)
- Files matching `*_test.lat` (e.g., `math_test.lat`, `parser_test.lat`)

Discovery is recursive, so tests nested in subdirectories are found too:

Listing 31.10: A test directory with a nested suite

```
tests/
  test_math.lat
  strings_test.lat
  integration/
    test_server.lat
```

Each discovered file gets its own header and its own tally, and the run ends with an aggregate line counted in *files*, not tests:

Listing 31.11: Running all tests in a directory

```
$ clat test tests/
Discovered 3 test files in tests/

Running tests in tests/integration/test_server.lat...
  FAIL  division by zero — expected error

Results: 0 passed, 1 failed (1 total)

Running tests in tests/strings_test.lat...
  PASS  multiplication
  PASS  floating-point arithmetic

Results: 2 passed, 0 failed (2 total)

Running tests in tests/test_math.lat...
  PASS  addition
  PASS  subtraction
  PASS  basic arithmetic

Results: 3 passed, 0 failed (3 total)

Overall: 2 files passed, 1 failed (3 total)
```

`Overall` Counts Files, Not Tests

`Overall: 2 files passed, 1 failed (3 total)` is talking about the three *files*. Six of the seven individual tests passed; one file contained the single failure, so that file is counted as failed. If you want a test-level total across a directory, add up the per-file `Results:` lines yourself.

31.2.2 Filtering Tests by Name

The `-f` or `--filter` flag runs only tests whose name contains the given substring:

Listing 31.12: Filtering tests by name

```
$ clat test tests/ -f "arithmetic"
Discovered 3 test files in tests/

Running tests in tests/integration/test_server.lat...
No tests matched filter "arithmetic" (1 skipped).

Running tests in tests/strings_test.lat...
  PASS  floating-point arithmetic

Results: 1 passed, 0 failed, 1 skipped (2 total)

Running tests in tests/test_math.lat...
  PASS  basic arithmetic

Results: 1 passed, 0 failed, 2 skipped (3 total)

Overall: 3 files passed, 0 failed (3 total)
```

The filter is a substring match (C's `strstr`, in `evaluator_run_tests()`). Non-matching tests are reported as `skipped` rather than silently dropped, and a file with nothing to run says so explicitly instead of printing an empty tally.

Notice that a filtered run exits 0 even though `tests/integration/test_server.lat` contains a failing test—the filter excluded it, so it never ran. That is what you want for a targeted debugging loop, and exactly what you don't want for a CI gate.

31.2.3 Verbose and Summary Modes

Two flags shape the output, both more modest than their names suggest.

`-v` (or `--verbose`) is accepted and plumbed all the way to `evaluator_set_test_verbose()`, but as of v0.6.4 the `test_verbose` field it sets is never read back in `src/eval.c`. Running `clat test -v` produces byte-identical output to running without it. The flag is wired for a feature that has not landed.

`--summary` controls the `Overall:` line, and only for directories. `src/main.c` prints that line when summary mode is on *or* more than one test file was discovered, so a directory holding two or more test files always gets it and `--summary` is redundant. Where the flag earns its keep is a directory that happens to contain exactly one test file: without it there is no aggregate line, with it there is—which keeps a CI log format stable as the suite grows from one file to many. Passing `--summary` alongside a single *file* path does nothing.

31.2.4 Full CLI Reference

Here is the complete set of `clat test` flags, straight from `clat test --help`:

Flag	Description
`-f, --filter <pattern>`	Only run tests whose name matches pattern
`-v, --verbose`	Verbose output
`--summary`	Show summary after run
`--gc-stress`	GC stress testing mode
`--no-regions`	Disable region-based memory
`--no-assertions`	Disable runtime assertions
`-h, --help`	Show help

Exit Codes

`clat test` exits with code 0 if all tests pass, and code 1 if any test fails. This makes it straightforward to integrate into CI pipelines and shell scripts. Remember the filter caveat from Section 31.2.2: a filtered run reports success for the tests it chose to run, so CI should invoke `clat test` without `-f`.

31.3 Assert Functions

Lattice provides a rich set of built-in assertion functions. These are not library functions—they're built into the evaluator and the runtime (`src/eval.c`, `src/runtime.c`), so they're always available without any imports.

31.3.1 `assert(condition, message?)`

The fundamental assertion. If the condition is falsy, the test fails with the optional message:

Listing 31.13: Basic assert usage

```
test "assert basics" {
    assert(1 == 1, "one equals one")
    assert(true, "true is truthy")
    assert(42, "non-zero integers are truthy")
    assert("hello", "non-empty strings are truthy")
}
```

When `assert` fails, it throws an error that the test runner catches. The failure line is your custom string verbatim, separated from the test name by an em dash:

Listing 31.14: A failing assert

```
test "this will fail" {
    let balance = -50
    assert(balance >= 0, "balance must not be negative")
}
```

Listing 31.15: What the runner prints

```
$ clat test failing.lat
  FAIL  this will fail — balance must not be negative

Results: 0 passed, 1 failed (1 total)
```

Omit the message and the assertion falls back to a bare `assertion failed`, which tells you which test broke but not why. The message argument is optional in the grammar and close to mandatory in practice.

31.3.2 `assert_eq(actual, expected)`

The workhorse of testing. Checks that two values are equal, and if not, shows both values in the failure message:

Listing 31.16: assert_eq with clear failure messages

```
test "string operations" {
    let greeting = "hello" + " " + "world"
    assert_eq(greeting, "hello world")

    let trimmed = "  spaces  ".trim()
    assert_eq(trimmed, "spaces")
}

test "array operations" {
    let numbers = [1, 2, 3, 4, 5]
    assert_eq(len(numbers), 5)
    assert_eq(numbers[0], 1)
    assert_eq(numbers[4], 5)
}
```

Drop the `.trim()` from that second assertion and the runner shows you both sides:

Listing 31.17: An assert_eq failure names both values

```
  FAIL  string operations — assert_eq: expected spaces, got   spaces
```

Note the argument order in the message: `expected` reports the *second* argument and `got` the first, matching the `assert_eq(actual, expected)` signature.

Note also that the values are printed unquoted, which matters when whitespace is the bug. The runner emits all four spaces, but nothing delimits them: the leading pair survives as a gap you have to squint at, and the trailing pair is invisible. An empty string against a whitespace-only string produces a failure line that looks like it is missing half its text. When you are chasing a discrepancy like that, put the delimiters in yourself:

Listing 31.18: Making whitespace visible in a failure message

```
test "string operations" {
    let trimmed = "  spaces  "
    assert_eq("[" + trimmed + "]", "[spaces]")
}
```

Listing 31.19: Now the boundaries are legible

```
  FAIL  string operations — assert_eq: expected [spaces], got [  spaces  ]
```

Otherwise this level of detail is worth its weight in gold during debugging. You don't need to add print statements—the assertion tells you what went wrong.

31.3.3 `assert_ne(actual, expected)`

The opposite of `assert_eq`. Fails if the two values are equal:

Listing 31.20: Asserting inequality

```
test "unique ID generation" {
    let id1 = generate_id()
    let id2 = generate_id()
    assert_ne(id1, id2)
}
```

31.3.4 `assert_true(value)` and `assert_false(value)`

Strict boolean checks. Unlike `assert`, which accepts any truthy value, `assert_true` expects exactly `true` and `assert_false` expects exactly `false`:

Listing 31.21: Strict boolean assertions

```
test "boolean checks" {
    let is_admin = check_role("admin")
    assert_true(is_admin)

    let is_expired = check_expiry("2099-12-31")
    assert_false(is_expired)
}
```

assert_true Is Strict

`assert_true(42)` will *fail* because `42` is not the boolean value `true`, even though it's truthy. The runner says so plainly:

```
  FAIL  assert_true strict — assert_true: expected true, got 42
```

If you want truthy checking, use `assert(42, "expected truthy")`.

31.3.5 assert_type(value, type_name)

Verifies that a value has the expected type. The type name is a string that matches the output of `typeof()`, such as `"Int"`, `"Float"`, `"String"`, `"Array"`, or the name of a struct:

Listing 31.22: Type assertions

```
struct Point {
    x: Float,
    y: Float
}

test "type checking" {
    assert_type(42, "Int")
    assert_type(3.14, "Float")
    assert_type("hello", "String")
    assert_type([1, 2, 3], "Array")
    assert_type(true, "Bool")

    let origin = Point { x: 0.0, y: 0.0 }
    assert_type(origin, "Point")
}
```

The `assert_type` function checks both the primitive type and, for structs, the struct name. So `assert_type(origin, "Point")` succeeds because the evaluator (in `src/eval.c`) compares the struct's name field against the expected string.

31.3.6 assert_contains(haystack, needle)

Checks that a string or array contains a given element:

Listing 31.23: Containment assertions

```
test "string containment" {
    let message = "the quick brown fox"
    assert_contains(message, "brown")
    assert_contains(message, "fox")
}

test "array containment" {
    let colors = ["red", "green", "blue"]
    assert_contains(colors, "green")
}
```

31.3.7 `assert_throws(closure)`

Verifies that a piece of code throws an error. Pass a closure that should fail:

Listing 31.24: Asserting that code throws an error

```
test "division by zero throws" {
    assert_throws(|_| {
        let result = 10 / 0
    })
}

test "index out of bounds throws" {
    assert_throws(|_| {
        let items = [1, 2, 3]
        let bad = items[99]
    })
}
```

The built-in `assert_throws` wraps the closure in a `try`/`catch` block. If no error is thrown, the assertion fails with `assert_throws: expected an error but none was thrown`.

Note the closure shape: `|_| { ... }`. `assert_throws` calls its argument with one argument, so the closure needs one parameter, and `_` is the conventional name for a parameter you intend to ignore. An empty closure would be `| |` with a space—`||` is the logical-or operator.

31.3.8 Summary of Built-in Assertions

Every built-in assertion prefixes its failure message with its own name, so the runner's output identifies the assertion that broke without you having to count lines. Here they are with the exact text each one produces:

Function	Failure message
`assert(cond, msg?)`	your `msg` verbatim, or `assertion failed`
`assert_eq(a, b)`	`assert_eq: expected <b>, got <a>`
`assert_ne(a, b)`	`assert_ne: values are equal: <a>`
`assert_true(v)`	`assert_true: expected true, got <v>`
`assert_false(v)`	`assert_false: expected false, got <v>`
`assert_type(v, t)`	`assert_type: expected type <t>, got <typeof(v)>`
`assert_contains(h, n)`	`assert_contains: <h> does not contain <n>`
`assert_throws(fn)`	`assert_throws: expected an error but none was thrown`

31.4 The Test Standard Library Module

Beyond the built-in assertions, Lattice ships with a test library at `lib/test.lat` that provides a more structured, framework-style testing experience. This is useful for larger projects that need test suites organized into groups.

31.4.1 Importing the Test Library

Listing 31.25: Importing the test library

```
import "lib/test" as t
```

Import Paths Resolve Against the Working Directory

That path is relative to the directory you run `clat` from, not to the file doing the importing. Run a script that says `import "lib/test"` from anywhere other than the Lattice source tree and every backend reports the same core message:

```
$ clat /tmp/scratch/suite.lat
vm error: import: cannot find 'lib/test.lat'
```

The prefix tracks the backend—`vm error:` on the default stack VM, plain `error:` under `--tree-walk`, `regvm error:` under `--regvm`—but the message is identical. Either `cd` into the tree first or give the import an absolute path.

31.4.2 Organizing Tests with `describe` and `it`

The test library provides two organizational primitives. The `describe()` function creates a named test suite, and `it()` creates individual test cases within a suite.

Listing 31.26: Structured test suites

```
import "lib/test" as t

t.run([
    t.describe("String operations", |_| {
        return [
            t.it("concatenation", |_| {
                t.assert_eq("hello" + " " + "world", "hello world")
            }),
            t.it("length", |_| {
                t.assert_eq(len("lattice"), 7)
            }),
            t.it("uppercase", |_| {
                t.assert_eq("hello".to_upper(), "HELLO")
            })
        ]
    }),
    t.describe("Math operations", |_| {
        return [
            t.it("addition", |_| {
                t.assert_eq(2 + 2, 4)
                t.assert_eq(1.5 + 2.5, 4.0)
            }),
            t.it("division by zero", |_| {
                t.assert_throws(|_| { 1 / 0 })
            })
        ]
    })
])
```

This is a program, not a test file, so you run it directly rather than through `clat test`. The output is identical on all three backends:

Listing 31.27: Structured test output

```
$ clat suite.lat
Running tests...

  String operations
    u2713 concatenation
    u2713 length
    u2713 uppercase

  Math operations
    u2713 addition
    u2713 division by zero

5 passed, 0 failed, 5 total
Completed in 0ms
```

Those `u2713` tokens are meant to be check marks. `lib/test.lat` writes the check mark as a backslash-`u2713` escape inside a `format()` string, and the Lattice lexer has no `\u` escape—so the backslash is dropped and the four remaining characters print literally. A failing case shows `u2717` (the intended ballot X) followed by the caught error message. Cosmetic, but you will see it, and it is a useful reminder that Lattice's string escapes are a short list.

`lib/test` Always Exits 0

The library's `run()` prints a tally and returns; it never sets a non-zero exit status. A suite where every case fails still exits 0, so a CI job that only checks the exit code will report green. For anything gating a merge, prefer built-in `test` blocks and `clat test`, which does exit 1 on failure (Section 31.2.4). If you want the library's structure *and* a usable exit code, have your runner script grep the output for `0 failed`.

Built-in Tests vs. the Test Library

The built-in `test "name" \{ ... \}` blocks are ideal for unit tests that live alongside source code, run by `clat test`, which reports failure through its exit code. The `lib/test` library is better for standalone test programs that need more structure—nested suites, timing, and a custom runner—and it is the only one of the two that runs on the backend of your choosing. Both approaches work well; use whichever fits your project.

31.4.3 Extended Assertions in the Test Library

The test library provides additional assertion functions beyond the built-ins:

Listing 31.28: Extended assertions from lib/test

```
import "lib/test" as t

t.run([
    t.describe("Comparisons", |_| {
        return [
            t.it("greater than / less than", |_| {
                t.assert_gt(10, 5)
                t.assert_lt(3, 7)
                t.assert_gte(5, 5)
                t.assert_lte(5, 5)
            }),
            t.it("floating-point proximity", |_| {
                let pi = 3.14159265
                t.assert_near(pi, 3.14, 0.01)
            }),
            t.it("nil checks", |_| {
                t.assert_nil(nil)
                t.assert_not_nil(42)
                t.assert_not_nil("hello")
            }),
            t.it("type checking", |_| {
                t.assert_type(42, "Int")
                t.assert_type("hi", "String")
            })
        ]
    })
])
```

Here's the full list of assertions provided by `lib/test.lat`:

Function	Description
`t.check(cond)`	Fail if condition is falsy
`t.assert_eq(actual, expected)`	Fail if values differ
`t.assert_neq(actual, expected)`	Fail if values are equal
`t.assert_gt(a, b)`	Fail if `a <= b`
`t.assert_lt(a, b)`	Fail if `a >= b`

Function	Description
`t.assert_gte(a, b)`	Fail if `a < b`
`t.assert_lte(a, b)`	Fail if `a > b`
`t.assert_near(actual, expected, epsilon)`	Fail if values differ by more than `epsilon`
`t.assert_contains(haystack, needle)`	Fail if haystack doesn't contain needle
`t.assert_throws(closure)`	Fail if the closure doesn't throw
`t.assert_type(value, type_name)`	Fail if the type doesn't match
`t.assert_nil(value)`	Fail if value is not `nil`
`t.assert_not_nil(value)`	Fail if value is `nil`
`t.assert_true(value)`	Fail if value is not `true`
`t.assert_false(value)`	Fail if value is not `false`

assert_near for Floating-Point Tests

Comparing floating-point numbers with `assert_eq` can lead to spurious failures due to rounding. Use `t.assert_near(actual, expected, epsilon)` instead, where `epsilon` is the acceptable margin of error. For most calculations, an epsilon of `0.0001` is a good starting point.

31.4.4 How the Test Library Works Internally

If you open `lib/test.lat`, you'll find that the library is written entirely in Lattice. There's no magic—it uses the built-in `assert()` function combined with `format()` and `repr()` to produce clear error messages, and `try`/`catch` to recover from assertion failures.

The `it()` function returns a `Map` with `"name"` and `"fn"` keys. The `describe()` function calls the builder closure to get an array of these Maps and wraps them in a suite Map with `"name"` and `"tests"` keys. The `run()` function iterates through suites and test cases, invoking each closure inside a `try`/`catch` block. If the closure completes without error, the test passes. If an assertion triggers an error, the catch block captures `e.message` and reports the failure.

The pattern is worth studying even if you never use the library, because it is a compact demonstration of `try` as an *expression*:

Listing 31.29: The heart of the runner, from lib/test.lat

```
let result = try {
    test_fn(nil)
    "pass"
} catch e {
    e.message
}
```

The `try` block yields `"pass"` when the closure returns normally and the caught message when it doesn't, collapsing success and failure into one string the loop can branch on. Timing comes from two `time()` calls bracketing the whole run—which is why short suites report `Completed in 0ms`.

31.5 Testing Strategies for Lattice Programs

Having the tools is one thing. Using them effectively is another. Let's explore strategies that experienced Lattice developers rely on.

31.5.1 Unit Testing: One Function, One Test

The most common pattern: write a test block for each public function. Keep tests close to the code they exercise.

Listing 31.30: Unit tests alongside the code

```
fn is_palindrome(s: String) -> Bool {
    let reversed = s.chars().reverse().join("")
    return s == reversed
}

test "is_palindrome with palindromes" {
    assert_true(is_palindrome("racecar"))
    assert_true(is_palindrome("level"))
    assert_true(is_palindrome("a"))
    assert_true(is_palindrome(""))
}

test "is_palindrome with non-palindromes" {
    assert_false(is_palindrome("hello"))
    assert_false(is_palindrome("lattice"))
}
```

31.5.2 Testing Edge Cases

The most productive tests often check boundaries: empty inputs, zero values, negative numbers, maximum sizes, and `nil`.

Listing 31.31: Testing edge cases

```
fn safe_divide(a: Float, b: Float) -> Float {
    if b == 0.0 {
        return 0.0
    }
    return a / b
}

test "safe_divide normal cases" {
    assert_eq(safe_divide(10.0, 2.0), 5.0)
    assert_eq(safe_divide(7.0, 3.5), 2.0)
}

test "safe_divide edge cases" {
    assert_eq(safe_divide(0.0, 5.0), 0.0)
    assert_eq(safe_divide(10.0, 0.0), 0.0)
    assert_eq(safe_divide(0.0, 0.0), 0.0)
    assert_eq(safe_divide(-10.0, 2.0), -5.0)
}
```

31.5.3 Testing Error Paths

Use `assert_throws` (either the built-in or the library version) to verify that functions fail when they should:

Listing 31.32: Testing that errors are thrown correctly

```
fn parse_positive_int(s: String) -> Int {
    let n = parse_int(s)
    if is_error(n) {
        return error("not a valid integer: " + s)
    }
    if n <= 0 {
        return error("expected positive integer, got " + to_string(n))
    }
    return n
}

test "parse_positive_int rejects invalid input" {
    assert_throws(|_| { parse_positive_int("abc") })
    assert_throws(|_| { parse_positive_int("-5") })
    assert_throws(|_| { parse_positive_int("0") })
}

test "parse_positive_int accepts valid input" {
    assert_eq(parse_positive_int("42"), 42)
    assert_eq(parse_positive_int("1"), 1)
}
```

31.5.4 Testing Structs and Methods

When testing structs, create instances with known values and verify field access, method results, and trait implementations:

Listing 31.33: Testing structs

```
struct Rectangle {
    width: Float,
    height: Float
}

fn area(r: Rectangle) -> Float {
    return r.width * r.height
}

fn perimeter(r: Rectangle) -> Float {
    return 2.0 * (r.width + r.height)
}

fn is_square(r: Rectangle) -> Bool {
    return r.width == r.height
}

test "rectangle area" {
    let r = Rectangle { width: 5.0, height: 3.0 }
    assert_eq(area(r), 15.0)
}

test "rectangle perimeter" {
    let r = Rectangle { width: 5.0, height: 3.0 }
    assert_eq(perimeter(r), 16.0)
}

test "square detection" {
    let square = Rectangle { width: 4.0, height: 4.0 }
    let rect = Rectangle { width: 4.0, height: 6.0 }
    assert_true(is_square(square))
    assert_false(is_square(rect))
}
```

31.5.5 Organizing a Test Suite

For a project of any size, keep tests in a dedicated `tests/` directory. Each test file mirrors the module it tests:

Listing 31.34: Project layout with tests

```
// src/
//   math.lat
//   strings.lat
//   http.lat
// tests/
//   test_math.lat
//   test_strings.lat
//   test_http.lat
```

Then run the entire suite with:

Listing 31.35: Running the full test suite

```
$ clat test tests/
```

31.5.6 The Test-First Workflow

A powerful practice: write the test *before* the implementation.

Listing 31.36: Step 1: write the test against a stub

```
fn fibonacci(n: Int) -> Int {
    // TODO: implement
    return 0
}

test "fibonacci sequence" {
    assert_eq(fibonacci(0), 0)
    assert_eq(fibonacci(1), 1)
    assert_eq(fibonacci(2), 1)
    assert_eq(fibonacci(5), 5)
    assert_eq(fibonacci(10), 55)
}
```

Listing 31.37: Step 2: run it, and watch it fail

```
$ clat test fib.lat
  FAIL  fibonacci sequence — assert_eq: expected 1, got 0

Results: 0 passed, 1 failed (1 total)
```

Step 3 is to implement until it passes.

Notice that the runner reported the *second* assertion, not the first: `fibonacci(0)` already returns 0, so the stub satisfies it by accident. A test block stops at its first failure—the loop in `evaluator_run_tests()` breaks out as soon as a statement errors—so a block with five assertions tells you about one problem per run. That is the right trade for a red-green loop, but it means a long test block hides how much work is left.

The failing test gives you a clear target. Once all assertions pass, you know the implementation is correct—at least for the cases you specified.

31.5.7 Testing with GC Stress Mode

The `--gc-stress` flag forces a collection at every allocation checkpoint instead of waiting for the heap to cross its threshold. In `src/eval.c` the collection guard tests the stress flag *or* the usual heap-threshold comparison, so the flag short-circuits the "is there enough garbage yet?" question and answers "collect now" every time.

Listing 31.38: Running tests with GC stress

```
$ clat test --gc-stress tests/
```

You can see the difference outside the test runner, where `--stats` is available. A loop that pushes two thousand strings onto an array reports `gc cycles: 0` on a normal run and `gc cycles: 2004` under `--gc-stress`—roughly one collection per allocation, exactly as advertised.

Under GC stress, any test that accidentally holds a dangling reference or forgets to root a value will fail quickly. It makes tests slower, so you wouldn't run this in every CI build—but running it periodically catches subtle bugs that normal execution misses.

31.5.8 Disabling Assertions for Performance Testing

The `--no-assertions` flag disables runtime assertions (specifically, the `debug_assert` built-in). This is useful when you want to benchmark code that includes debug assertions without removing them from the source:

Listing 31.39: Running without runtime assertions

```
$ clat test --no-assertions benchmarks/
```

Note that `assert()` (the testing assertion) is not affected by this flag—only `debug_assert()` is suppressed. A test whose only content is a failing `debug_assert` fails normally and *passes* under `--no-assertions`, which is a compact way to check that the flag is doing what you think:

Listing 31.40: A test that changes verdict with the flag

```
test "debug assert" {
    debug_assert(1 == 2, "debug assert fired")
}
```

Listing 31.41: Same file, two verdicts

```
$ clat test dbgassert.lat
  FAIL  debug assert — debug assert fired

Results: 0 passed, 1 failed (1 total)

$ clat test --no-assertions dbgassert.lat
  PASS  debug assert

Results: 1 passed, 0 failed (1 total)
```

Don't Let `--no-assertions` into CI

That second run is a green suite for a test that asserts `1 == 2`. The flag is a benchmarking tool: it strips the checks so their cost doesn't pollute your timings. Used on a real suite it quietly disables every `debug_assert` in the codebase under test.

Exercises

1. Write a `stack.lat` module that implements a stack using an array (with `push`, `pop`, `peek`, and `is_empty` functions). Include inline `test` blocks that verify each operation, including edge cases like popping from an empty stack.
2. Create a structured test suite using `lib/test` that tests a string utility library with at least three `describe` groups and five total test cases. Use `assert_eq`, `assert_contains`, and `assert_throws`.
3. Write a test file that uses `assert_near` to verify floating-point calculations (trigonometry, compound interest, or physics formulas). Experiment with different epsilon values to understand how precision affects test results.
4. Set up a small project with three source files in `src/` and corresponding test files in `tests/`. Run the entire test suite, then use `--filter` to run only the tests related to one module. Write a one-line CI command that formats and tests the whole project.
5. Write a test that verifies the *behavior* of the phase system. Freeze an array, assert that `phase_of` now reports `"crystal"`, wrap a `push` in `assert_throws` to confirm the mutation is refused, and assert the length is unchanged. Then write the mirror-image test for `thaw`.
6. Take one of your own suites and run it twice: once normally, once with `--no-assertions`. If the results differ, you have found a test that depends on `debug_assert`. Decide whether that assertion belongs in the code under test or in the test itself.

What's Next

When a test fails and the assertion message isn't enough to pinpoint the problem, you need a more powerful tool: a debugger. In Chapter 32, we'll explore Lattice's built-in CLI debugger, learn how to set breakpoints, step through code line by line, inspect variables, watch expressions change, and drive the whole thing from an editor over the Debug Adapter Protocol. We'll also map out which of those capabilities belong to which backend—because unlike `clat test`, the debugger is particular about where it runs.

Chapter 32

Debugging

Tests tell you *that* something broke. The debugger tells you *why*. Lattice includes a built-in interactive debugger that lets you pause execution, inspect variables, step through code line by line, and evaluate expressions—all without leaving the terminal. The same machinery also speaks the Debug Adapter Protocol (DAP), so an editor can drive it over stdin/stdout. And for the trickiest bugs—the ones where a value changed "somewhere" but you don't know where—Lattice's value history functions let you track mutations over time.

This chapter comes with a map of the terrain, because the debugging tools are the least uniform corner of Lattice. Some of them exist on one backend only; one of them will end your program if you type the wrong thing at its prompt. Knowing which is which turns a frustrating afternoon into a productive one, so let's put the map first.

The Debugger Runs on the Default Stack VM Only

`--debug`, `--break`, and `--dap` are attached in the stack-VM execution path in `src/main.c` and nowhere else. Combine any of them with `--tree-walk` or `--regvm` and the flag is *silently ignored*—no warning, no prompt, the program runs to completion:

```
$ clat --tree-walk --debug orders.lat
49.75
```

Flag order makes no difference. If you reach for the debugger and get output instead of a `(dbg)` prompt, check whether a backend flag is still on the command line.

32.1 The –debug Flag and the CLI Debugger

To start a program under the debugger, add the `--debug` flag. We'll use this small program throughout the chapter:

Listing 32.1: tax.lat, our specimen

```
fn add_tax(price: Float, rate: Float) -> Float {
    let tax = price * rate
    return price + tax
}

let total = add_tax(100.0, 0.08)
print(total)
```

Listing 32.2: Launching the debugger

```
$ clat --debug tax.lat
Lattice debugger attached.
Stepping from first instruction. Type 'help' for commands.

Stopped at line 1
-->    1 | fn add_tax(price: Float, rate: Float) -> Float {

(dbg)
```

The program pauses at the very first line, with `-->` marking where execution sits. You're now in the *debug REPL*—an interactive prompt where you can inspect state, set breakpoints, and control execution.

The Debug REPL

The debug REPL is a command loop that activates whenever execution is paused—either at startup in step mode, at a breakpoint, or after a step command completes. The prompt `(dbg)` indicates that the program is paused and waiting for your instruction. All debugger commands are processed by the `debugger_check()` function in `src/debugger.c`.

The Debugger Talks on stderr

Every line the debugger prints—banner, `(dbg)` prompt, source context, variable values—goes to standard error. Your program's `print()` output goes to standard out. That separation is what makes `clat --debug prog.lat > run.log` useful: the log holds the program's output and the terminal keeps the debugging conversation. It also means that redirecting stderr away hides the prompt while the debugger keeps waiting for input, which looks exactly like a hang.

32.1.1 The Help Command

Type `help` (or `h`) at the debug prompt to see all available commands:

Listing 32.3: Debugger help output

```
(dbg) help
Debugger commands:
  s, step           Step into (execute one line, enter calls)
  n, next           Step over (execute one line, skip calls)
  o, out, finish    Step out (run until current function returns)
  c, continue       Continue until next breakpoint
  l, locals         Show local variables
  bt, stack         Show call stack / backtrace
  list              Show source context around current line
  b <line|func>     Set breakpoint at line or function
  b <N> if <expr>   Set conditional breakpoint
  d <id>            Delete breakpoint by ID
  p <var>           Print variable value
  eval <expr>       Evaluate expression
  watch <expr>      Add watch expression
  unwatch <id>      Remove watch expression
  info b            List breakpoints
  info w            List watch expressions
  q, quit           Exit debugger
  h, help           Show this help
```

32.1.2 Initial Breakpoint with –break

If you know roughly where the bug is, you can jump straight to a specific line instead of stepping from the beginning. Here it is against a loop that accumulates a running sum:

Listing 32.4: gloop.lat

```
flux index = 0
flux running_sum = 0

while index < 5 {
    running_sum = running_sum + index * 10
    index = index + 1
}

print(running_sum)
```

Listing 32.5: Starting with a breakpoint

```
$ clat --break 5 gloop.lat
Lattice debugger attached.
Breakpoints: line 5

Breakpoint 1 at line 5
-->    5 |     running_sum = running_sum + index * 10

(dbg)
```

The `--break` flag implies `--debug`, and it also flips the debugger out of step mode into continue mode—which is why the program runs at full speed to line 5 instead of pausing at line 1.

32.1.3 Viewing Source Context

The `list` command shows the source code surrounding the current line, with an arrow pointing to where execution is paused:

Listing 32.6: Viewing source context

```
(dbg) list
       1 | fn add_tax(price: Float, rate: Float) -> Float {
-->    2 |     let tax = price * rate
       3 |     return price + tax
       4 | }
       5 |
       6 | let total = add_tax(100.0, 0.08)
       7 | print(total)
```

The debugger loads the whole source file at startup (via `debugger_load_source()` in `src/debugger.c`) so it can display context around the current execution point. The window is five lines either side of the current one—a seven-line file like `tax.lat` fits entirely, while stopping at line 25 of a forty-line file shows lines 20 through 30.

32.1.4 Inspecting Variables

The `p` command shows the value of a variable. Stop inside `add_tax` and both parameters are there for the asking:

Listing 32.7: Printing a variable

```
(dbg) p price
price = 100

(dbg) p rate
rate = 0.08
```

The `locals` command (or `l`) shows *all* named locals in the current frame:

Listing 32.8: Viewing all local variables

```
(dbg) locals
  price = 100
  rate = 0.08
  tax = 100
```

Under the hood, the debugger reads local variable names from the chunk's `local_names` array and their values from the VM's stack slots. If a variable has no debug name (e.g., a compiler-generated temporary), it won't appear in the listing. At the top level there are no frame locals at all, and `locals` says `(no local variable info)`.

Locals Are Shown Before Their Line Runs

That `tax = 100` in the listing above is a lie of timing, not of reporting. The debugger pauses *before* executing the highlighted line, so at line 2 the slot that will hold `tax` has not been assigned yet and still contains whatever the VM left there. Step once more and `p tax` reports the real `8`. When a local looks impossible, check whether its assignment has actually run.

32.1.5 Evaluating Expressions

The `eval` command evaluates a Lattice expression against the paused program. We need a second specimen with a global and a local of the same shape, so meet `orders.lat`, which we'll keep using for the rest of the CLI sections:

Listing 32.9: orders.lat

```
fn calculate_total(items: Array) -> Float {
    flux total = 0.0
    for price in items {
        total = total + price
    }
    return total
}

let prices = [12.5, 30.0, 7.25]
let total = calculate_total(prices)
print(total)
```

Break at line 4 and `eval` happily handles arithmetic, builtin calls, and the top-level `prices` binding:

Listing 32.10: Evaluating expressions in the debugger

```
(dbg) eval 1 + 1
=> 2

(dbg) eval len("abc")
=> 3

(dbg) eval prices
=> [12.5, 30, 7.25]
```

You can call functions, perform arithmetic, index into containers—anything you could write in a normal Lattice expression, with one large exception that we need to take seriously.

`eval` Cannot See Locals—and Failing Is Not Free

`debugger_eval_expr()` lexes, parses, and compiles your text as a fresh top-level chunk with `stack_compile_repl()`, then runs that chunk on the live VM. A fresh top-level chunk resolves *globals*; function locals live in stack slots it knows nothing about. Paused at line 4 of `orders.lat` there are two things called `total`—the loop's local and the not-yet-assigned global—and the two commands disagree about which one exists:

```
(dbg) p total
total = 0

(dbg) eval total
Error: undefined variable 'total'
```

The prompt comes back and the session looks healthy. It is not. On the next `c` the program dies:

```
(dbg) c
vm error: function 'calculate_total' return type expects Float, got Unit
stack trace (most recent call last):
  [line 10] in <script>
  [line 1] in calculate_total()
```

A *failed* evaluation leaves the paused program in a state it cannot recover from. A successful one (`eval prices`, `eval 1 + 1`) is far gentler—the program's data is intact and it runs to completion—though it does cost one repeated stop, as Section 32.4.1 explains. Reach for `p <name>` and `locals` when you want a local—they read the frame slots directly and always work—and save `eval` for globals and self-contained expressions.

The same warning applies to evaluation you didn't type yourself: breakpoint conditions and watch expressions go through `debugger_eval_expr()` too, so both inherit the globals-only rule. Sections 32.2.3 and 32.4.1 show what that looks like in practice.

Side Effects in Eval

Expressions evaluated with `eval` run on the real VM state, and mutations persist. Given a global `flux items = [1, 2, 3]` and a breakpoint on the line that prints it:

```
(dbg) eval items.push(99)
=> ()

(dbg) eval items
=> [1, 2, 3, 99]

(dbg) c
[1, 2, 3]
4
```

The push took effect—but look at what the program then printed. `print(items)` reports the pre-push array while `print(items.len())` on the next line reports 4. The two disagree because the re-entrant evaluation disturbed the very statement that was paused mid-flight.
Use `eval` for inspection, not mutation. When you truly need to perturb a running program, change the source and re-run—you will spend less time than you will unpicking a contradiction like the one above.

32.1.6 The Call Stack

The `bt` (or `stack` or `backtrace`) command shows the complete call chain:

Listing 32.11: Viewing the call stack

```
(dbg) bt
Backtrace:
  #0 [line 6] in <script>
  #1 [line 2] in add_tax  <-- current
```

Frame `#0` is the bottom of the stack (the top-level script), and the frame marked `<-- current` is where execution is paused. Each frame reports the line it is currently *on*, so `#0` shows line 6—the call site—rather than the line the script started at. This helps you understand how you got to the current line, especially when the same function is called from several places.

32.2 Breakpoints: Line, Function, Conditional

Breakpoints tell the debugger "stop here." Lattice supports three kinds: line breakpoints, function breakpoints, and conditional breakpoints.

32.2.1 Line Breakpoints

Set a breakpoint at a specific line number:

Listing 32.12: Setting a line breakpoint

```
(dbg) b 4
Breakpoint 1 set at line 4

(dbg) c

Breakpoint 1 at line 4 in calculate_total()
-->    4 |         total = total + price

(dbg)
```

The debugger assigns each breakpoint an auto-incrementing ID (starting at 1). When execution hits the breakpoint, the debugger reports the breakpoint number, the line, and—when the line is inside a function—which function it belongs to.

32.2.2 Function Breakpoints

If you don't know the line number but know the function name, set a function breakpoint:

Listing 32.13: Setting a function breakpoint

```
(dbg) b calculate_total
Breakpoint 1 set on function calculate_total()

(dbg) c

Breakpoint 1 at calculate_total(), line 1 in calculate_total()
-->    1 | fn calculate_total(items: Array) -> Float {

(dbg)
```

Function breakpoints trigger whenever the named function is entered, regardless of which line called it. The debugger detects function entry by comparing the current frame count against the previous check—if the frame count increased and the new frame's function name matches, the breakpoint fires (see the function breakpoint check in `debugger_check()`).

32.2.3 Conditional Breakpoints

A conditional breakpoint only pauses when a Lattice expression evaluates to a truthy value. The syntax is `b <line> if <expression>`, and function breakpoints take conditions the same way.

Here it is on `gloop.lat`, stopping on the fourth trip through the loop and nowhere else:

Listing 32.14: A conditional breakpoint that fires once

```
(dbg) b 5 if index == 3
Breakpoint 1 set at line 5 if index == 3

(dbg) c

Breakpoint 1 at line 5
-->    5 |     running_sum = running_sum + index * 10

(dbg) p index
index = 3

(dbg) p running_sum
running_sum = 30
```

The condition is evaluated every time the breakpoint location is reached. If the condition is falsy, execution continues without pausing.

Conditions Are Globals-Only Too

Breakpoint conditions run through `debugger_eval_expr()`, so they inherit the restriction from Section 32.1.5: a condition naming a function *local* cannot resolve it. The debugger tells you, then stops anyway so you can fix the condition:

```
(dbg) b 4 if price > 20.0
Breakpoint 1 set at line 4 if price > 20.0

(dbg) c
Warning: breakpoint 1 condition error: undefined variable 'price'
  (did you mean 'prices'?)

Breakpoint 1 at line 4 in calculate_total()
-->    4 |         total = total + price
```

Breaking on error is the safe default—a silently-never-firing breakpoint would be worse—but it means a condition on a local degrades into an unconditional breakpoint. The condition above works perfectly in `gloop.lat` because `index` is a top-level binding. When you need to filter on a local, hoist the value into a top-level `flux` variable for the duration of the hunt, or use `breakpoint()` (Section 32.4.2) with an ordinary `if` around it.

Use Conditional Breakpoints for Loops

If you're debugging a loop that runs thousands of times but only the 500th iteration is wrong, a conditional breakpoint is far more effective than stepping through all 500 iterations—provided the counter is a global:

```
(dbg) b 20 if index == 500
```

32.2.4 Managing Breakpoints

List all breakpoints with `info b` (or `info breakpoints`):

Listing 32.15: Listing breakpoints

```
(dbg) info b
Num  Type      Enb  What
1    line      y    line 4  (hits: 1)
2    function  y    calculate_total()  (hits: 0)
3    line      y    line 5  if index == 3  (hits: 0)
```

The table shows each breakpoint's ID, type (line or function), whether it's enabled (`y`/`n`), what it's set on, any condition, and how many times it has been hit. With none set, `info b` answers `No breakpoints set.`

Remove a breakpoint by its ID:

Listing 32.16: Deleting a breakpoint

```
(dbg) d 2
Breakpoint 2 removed
```

A Breakpoint in a Loop Stops Once Per Iteration

Repeatedly continuing through `b 5` in `gloop.lat` stops five times—`p index` reports 0, 1, 2, 3, 4—and then the program finishes. That is the behavior to expect, and the baseline against which the stall described in Section 32.4.1 is measured.

32.3 Step-Into, Step-Over, Step-Out

The three stepping commands are the bread and butter of interactive debugging. They control how much code executes before the debugger pauses again.

32.3.1 Step-Into (s)

The `s` (or `step`) command executes one line. If that line contains a function call, the debugger enters the function and pauses at its first line. Stepping through `tax.lat` from the top:

Listing 32.17: Stepping into a function call

```
Stopped at line 6
-->    6 | let total = add_tax(100.0, 0.08)

(dbg) s

Stopped at line 1 in add_tax()
-->    1 | fn add_tax(price: Float, rate: Float) -> Float {

(dbg) s

Stopped at line 2 in add_tax()
-->    2 |     let tax = price * rate

(dbg) p price
price = 100

(dbg) s

Stopped at line 3 in add_tax()
-->    3 |     return price + tax

(dbg) p tax
tax = 8
```

Two details worth noticing. Entering a function stops on its *signature* line first, not on its first statement—so calling into a function costs two steps, not one. And once inside, the header changes from a bare `Stopped at line N` to `Stopped at line N in add_tax()`, which is the quickest way to tell at a glance whether you are in a frame or at the top level.

Step-into follows the actual execution path. If a line calls multiple functions, stepping enters the first one called.

32.3.2 Step-Over (n)

The `n` (or `next`) command executes one line but treats function calls as single steps—it doesn't enter them:

Listing 32.18: Stepping over a function call

```
Stopped at line 6
-->    6 | let total = add_tax(100.0, 0.08)

(dbg) n

Stopped at line 6
-->    6 | let total = add_tax(100.0, 0.08)

(dbg) n

Stopped at line 7
-->    7 | print(total)

(dbg) p total
total = 108
```

The function `add_tax` executed fully, but the debugger didn't pause inside it. The first `n` leaves us on line 6 a second time—the call has returned and the assignment half of the line is still ahead—and the second `n` moves on. Step-over is the right choice when you trust the called function and want to see its result.

The implementation tracks the call depth: when `next` is issued, the debugger records the current frame count and only pauses when the frame count is at or below that level *and* the line has changed (see the `next_mode` and `next_depth` fields in the `Debugger` struct).

32.3.3 Step-Out (o)

The `o` (or `out` or `finish`) command runs until the current function returns, then pauses in the caller:

Listing 32.19: Stepping out of a function

```
Stopped at line 3 in add_tax()
-->    3 |     return price + tax

(dbg) o

Stopped at line 6
-->    6 | let total = add_tax(100.0, 0.08)
```

Step-out lands you back on the *call site*, line 6, with the assignment still pending—`p total` there reports `variable 'total' not found`, because the binding does not exist yet. One more `n` completes the line and `total` appears.

Step-out is perfect when you've accidentally stepped into a function you don't care about. Rather than stepping through every line, `o` runs the rest of the function at full speed and returns you to the caller.

32.3.4 Continue (c)

The `c` (or `continue`) command resumes execution at full speed until the next breakpoint is hit (or the program ends):

Listing 32.20: Continuing execution

```
(dbg) b 4
Breakpoint 1 set at line 4

(dbg) c

Breakpoint 1 at line 4 in calculate_total()
-->    4 |         total = total + price

(dbg)
```

With no breakpoints left to hit, `c` runs the program to completion and the debugger exits with it.

Choosing the Right Step Command

- **Step-into (s)**: Use when you want to see what happens *inside* a function call.
- **Step-over (n)**: Use when you trust the called function and want to move to the next line.
- **Step-out (o)**: Use when you're inside a function you don't care about and want to return to the caller.
- **Continue (c)**: Use when you've set breakpoints and want to skip ahead.

32.4 Watch Expressions and `breakpoint()`

32.4.1 Watch Expressions

A watch expression is monitored automatically every time the debugger pauses. Instead of typing `p running_sum` at every stop, add a watch:

Listing 32.21: Adding watch expressions

```
(dbg) watch running_sum
Watch 1: running_sum

(dbg) watch index * 2
Watch 2: index * 2
```

Now every time the debugger pauses, it prints the current value of each watch expression:

Listing 32.22: Watch expressions displayed at each stop

```
(dbg) c

Breakpoint 1 at line 5
-->    5 |     running_sum = running_sum + index * 10

Watch expressions:
  [1] running_sum = 0
  [2] index * 2 = 0

(dbg)
```

Watch expressions can be any valid Lattice expression—not only variable names. Because they run through `debugger_eval_expr()`, the globals-only rule from Section 32.1.5 applies here too; a watch on a name that isn't in scope reports `<error: undefined variable 'name'>` in place of a value rather than failing the command.

Manage watches with `info w` and `unwatch`:

Listing 32.23: Managing watch expressions

```
(dbg) info w
  [1] running_sum = 0
  [2] index * 2 = 0

(dbg) unwatch 2
Watch 2 removed
```

An Active Watch Stalls the Program (v0.6.4)

Watches are the one debugger feature to use sparingly right now, because on the default stack VM an active watch prevents the program from making progress.

Every expression evaluation—interactive `eval`, a breakpoint condition, or a watch—compiles a fresh chunk and runs it on the live VM, and that costs the paused program its position: the next `c` lands on the same instruction instead of advancing. One `eval` costs one repeated stop and nothing more. A watch re-evaluates at *every* stop, so the loss repeats forever.

The arithmetic is visible on `gloop.lat`, whose loop runs five times. With a breakpoint on line 5 and no watches, `c` stops five times—`p index` walking 0, 1, 2, 3, 4—and the program prints `100`. Add `watch running_sum` first and `p index` reports `0` at every stop, no matter how many times you continue; the program never reaches its `print`. The expression being watched is irrelevant: `watch 1 + 1` stalls it as thoroughly.

`unwatch` restores forward motion. After removing the watch, one more `c` is absorbed paying off the last stall, and the next one advances normally.

Until this is fixed, prefer `p <name>` and `locals` at each stop—they read frame slots directly, never compile anything, and never disturb the program.

32.4.2 The `breakpoint()` Built-in

Sometimes the condition for pausing is too complex to express in a breakpoint condition string—or, as we saw in Section 32.2.3, it depends on a local the condition evaluator cannot reach. For these cases, Lattice provides the `breakpoint()` built-in function. Call it anywhere in your code, guard it with an ordinary `if`, and the interpreter drops into a small prompt when it gets there:

Listing 32.24: Using breakpoint() in code

```
fn total_of(items: Array) -> Float {
    flux total = 0.0
    for item in items {
        total = total + item.get("price")

        // Pause if we encounter a suspiciously high price
        if item.get("price") > 10000.0 {
            breakpoint()
        }
    }
    return total
}
```

Run that against an order containing a 24,000-unit line item and the program stops itself:

Listing 32.25: The breakpoint() prompt

```
$ clat bporder.lat

-- breakpoint() hit at line 13 --
Commands: locals, bt, p <var>, <expr>, cont

(breakpoint) locals
  items = [{"name": stapler, "price": 12}, {"name": server rack, "price": 24000}]
  total = 24012
  item = {"name": server rack, "price": 24000}
(breakpoint) p total
total = 24012
(breakpoint) cont
24012
```

`breakpoint()` Is Not a No-Op Without `--debug`

Notice that there is no `--debug` on that command line. `breakpoint()` does not check whether a debugger is attached—it opens its prompt on any ordinary run and blocks waiting for input on standard in. Left in a batch job, a service, or a CI step, it will hang the process until stdin closes. This is not a debug-only annotation you can safely forget about; treat every `breakpoint()` call as code you must remove before shipping.

`breakpoint()` Behaves Differently on Every Backend

`breakpoint()` is the sharpest divergence in the language, and it is worth knowing all three shapes before you rely on it.

Three Backends, Three Behaviors

Default stack VM. Opens the prompt, tagged with the line number: `-- breakpoint() hit at line 13 --`. `locals`, `bt`, `p <var>`, and `cont` all work. Typing anything else is treated as an expression, and *any* expression ends the run. A bare local name doesn't even resolve first:

```
(breakpoint) total
error: undefined variable 'total'
$
```

Even a self-contained expression, which does produce an answer, is terminal:

```
(breakpoint) 1 + 1
=> 2
$
```

The prompt's evaluator calls `stackvm_run()` re-entrantly on the same VM (see `native_breakpoint` in `src/runtime.c`), which unwinds the frames of the program that was paused.

`--tree-walk`. Opens the prompt with no line number—a bare `-- breakpoint() hit --`—and resolves bare names, so expressions work and the run continues afterwards:

```
(breakpoint) total
=> 24012
(breakpoint) total + 1
=> 24013
(breakpoint) cont
24012
```

Its `locals` is weaker, though: at a breakpoint inside a function it answers `(no locals in current scope)`, and its `bt` prints frames without line numbers (`#0 in total_of`). `p total` still works.

`--regvm`. Declines entirely, prints one line to stderr, and keeps running:

```
breakpoint(): not supported under the register VM backend
24012
```

The register VM stores frames in a different structure, and `native_breakpoint` refuses the cast rather than risk type confusion.

The practical rule: on the default VM, use `locals`, `p`, `bt`, and `cont` and nothing else. If you genuinely need to evaluate expressions against a paused frame, re-run the same program under `--tree-walk`, where the prompt is a real REPL over the live environment.

Programmatic Breakpoints

`breakpoint()` is especially useful for catching rare conditions that are hard to reproduce, and it is the only pausing mechanism that can filter on a function local. Rather than guessing which line to break on, embed the breakpoint directly in the condition logic:

```
if is_error(result) {
    breakpoint()  // pause only when something went wrong
}
```

32.5 DAP and VS Code Integration

The CLI debugger is powerful, but sometimes you want a graphical view: variable panes, inline source highlighting, and click-to-set breakpoints. Lattice ships a Debug Adapter Protocol (DAP) server for exactly that—the same protocol VS Code, Neovim (via nvim-dap), and Emacs (via dap-mode) speak.

32.5.1 What Is DAP?

The Debug Adapter Protocol, developed by Microsoft for VS Code, defines a JSON-based message protocol between an editor (the "client") and a debugger (the "adapter"). Instead of every editor implementing custom support for every language's debugger, DAP provides a common interface. Lattice's DAP implementation lives in `src/dap.c` and speaks DAP over stdin/stdout using Content-Length framed JSON messages.

32.5.2 Starting DAP Mode

To launch the debugger in DAP mode:

Listing 32.26: Launching DAP mode

```
$ clat --dap server.lat
```

Nothing appears—the process is waiting for a client, and its stdout is now the protocol channel rather than a terminal. The handshake follows the DAP specification: the client sends `initialize`, `launch`, optionally `setBreakpoints` and `setFunctionBreakpoints`, and finally `configurationDone`.

Because the transport is plain framed JSON on two pipes, you can drive it by hand to see what the adapter actually does. Sending `initialize` gets this back:

Listing 32.27: The initialize response, captured

```
Content-Length: 325

{"seq":1,"type":"response","request_seq":1,"success":true,
 "command":"initialize","body":{
   "supportsConfigurationDoneRequest":true,
   "supportsFunctionBreakpoints":true,
   "supportsConditionalBreakpoints":true,
   "supportsEvaluateForHovers":true,
   "supportsStepBack":false,
   "supportsSetVariable":false,
   "supportsSteppingGranularity":false}}
```

32.5.3 DAP Capabilities

That response is the definitive capability list (assembled by `dap_handshake()` in `src/dap.c`). The three `false` entries are as informative as the four `true` ones, so here is the whole table:

Capability	v0.6.4	Meaning
`supportsConfigurationDoneRequest`	yes	Client signals when breakpoint setup is complete
`supportsFunctionBreakpoints`	yes	Breakpoints on function names
`supportsConditionalBreakpoints`	yes	Breakpoint conditions
`supportsEvaluateForHovers`	yes	Expression evaluation on editor hover
`supportsStepBack`	no	No reverse execution
`supportsSetVariable`	no	The Variables pane is read-only
`supportsSteppingGranularity`	no	Stepping is per line, never per instruction

`supportsSetVariable: false` is the one to remember day to day: you can inspect anything in the Variables pane, but you cannot edit a value and resume with it changed.

32.5.4 What the Adapter Sends

A session against `orders.lat` with a breakpoint on line 4 shows every piece an editor UI is built from. The `stackTrace` response carries frames newest-first, each with a resolved source path:

Listing 32.28: stackTrace

```
{"stackFrames":[
  {"id":1,"name":"calculate_total","line":4,"column":1,
   "source":{"path":"/tmp/orders.lat","name":"orders.lat"}},
  {"id":0,"name":"<script>","line":10,"column":1,
   "source":{"path":"/tmp/orders.lat","name":"orders.lat"}}]}
```

`scopes` offers two, and marks the global one expensive so the editor fetches it lazily:

Listing 32.29: scopes

```
{"scopes":[
  {"name":"Locals","presentationHint":"locals",
   "variablesReference":1,"expensive":false},
  {"name":"Globals","presentationHint":"globals",
   "variablesReference":2,"expensive":true}]}
```

`variables` on the Locals reference returns the frame slots, typed, with a non-zero `variablesReference` on anything expandable:

Listing 32.30: variables

```
{"variables":[
  {"name":"items","value":"[12.5, 30, 7.25]","type":"Array",
   "variablesReference":1000},
  {"name":"total","value":"0","type":"Float","variablesReference":0},
  {"name":"price","value":"12.5","type":"Float","variablesReference":0}]}
```

And your program's output arrives as events rather than on the pipe directly:

Listing 32.31: An output event

```
{"type":"event","event":"output",
 "body":{"category":"stdout","output":"49.75\n"}}
```

Compound Variable Expansion

When the Variables pane shows a struct, array, or map, the editor can expand it to show individual fields or elements. This works through DAP's variable reference system: each compound value gets a reference ID (the `1000` above), and expanding it sends a `variables` request carrying that reference. The implementation in `src/dap.c` uses a per-stop variable reference table, cleared on each stop event to avoid stale data—so a reference held across a `continue` comes back as an empty list rather than yesterday's values.

`evaluate` Inherits the Locals Restriction

The `evaluate` request routes to the same `debugger_eval_expr()` as the CLI `eval` command, so Section 32.1.5 applies unchanged. Asking for a global succeeds; asking for a function local fails:

```
{"success":false,"command":"evaluate",
 "message":"undefined variable 'total'"}
```

And as in the CLI, the failure is not free: after it, continuing the session ends the program with `vm error: function 'calculate_total' return type expects Float, got Unit`. In an editor this looks like the debugger crashing your program for no reason—the reason is the hover or Debug Console expression you just evaluated. Read locals from the Variables pane, which reads frame slots directly, and keep the Debug Console for globals.

32.5.5 Connecting an Editor

The Bundled VS Code Extension Has No Debugger Contribution

The extension in `editors/vscode/` is version 0.2.0 and contributes `languages`, `grammars`, `configuration`, and `commands`—syntax highlighting and language-server features. It does *not* contribute a `debuggers` section, and nothing in the repository registers a `lattice` debug type. A `launch.json` naming `"type": "lattice"` therefore has nothing to bind to.

The adapter itself is real and works; what's missing is the thin VS Code extension that would launch `clat --dap` and register the type. Until one ships, use a DAP client you configure directly—nvim-dap, dap-mode, or your own harness.

Neovim's nvim-dap takes the adapter as an executable, which is all Lattice needs:

Listing 32.32: Neovim nvim-dap configuration (Lua)

```lua
local dap = require("dap")
dap.adapters.lattice = {
    type = "executable",
    command = "clat",
    args = { "--dap" },
}
dap.configurations.lattice = {{
    name = "Debug Lattice",
    type = "lattice",
    request = "launch",
    program = "${file}",
    stopOnEntry = true,
}}
```

Note the `args`: `{ "--dap" }` and not `{ "run", "--dap" }`, because there is no `run` subcommand—`clat` takes the filename directly.

`stopOnEntry = true` tells the adapter to pause at the first line, the DAP equivalent of `--debug` in CLI mode; set it `false` and the program runs until it hits a breakpoint.

Once connected, an editor built on those messages gives you the familiar experience:

- **Variables pane**: Locals and Globals as separate scopes, with arrays, structs, and maps expandable. Read-only—see `supportsSetVariable` above.
- **Call stack**: the `stackTrace` frames, from the current function down to `<script>`.
- **Breakpoint gutter**: click-to-set line breakpoints, plus conditions and function breakpoints, all advertised as supported.
- **Step controls**: step-into, step-over, step-out, and continue, at line granularity.
- **Debug console**: expression evaluation, with the locals caveat above.
- **Output capture**: `print()` output arrives as `output` events tagged `stdout`.

32.6 Value History: `track()`, `history()`, `rewind()`

Some bugs aren't about the current state—they're about how the state got there. A variable has the wrong value, but when did it change? What was it before? Lattice's value history functions let you answer these questions without setting breakpoints on every assignment.

32.6.1 `track(variable)` — Start Recording

The `track()` function tells the runtime to record every value change for a named variable:

Listing 32.33: Tracking a variable

```
flux counter = 0
track(counter)

counter = 10
counter = 20
counter = 30
```

After calling `track(counter)`, every subsequent assignment to `counter` is recorded as a snapshot. The snapshot includes a deep copy of the value, its phase, and the source line number.

track(variable)

`track(variable)` enables value history tracking for the specified variable. The argument is a variable name (not a string—the compiler automatically converts it to a string behind the scenes). Tracking begins with an initial snapshot of the variable's current value. Subsequent mutations are recorded as they happen.

Tracking Works on All Three Backends—With Caveats

Unlike the debugger, `track` is a runtime facility and works everywhere. The recording quality is not identical, though, and two differences will trip you up if you don't expect them:

- `--tree-walk` records every snapshot with `line: 0`. The values and their order are right; the line numbers are absent.
- `--regvm` records assignments to *globals* faithfully, but inside a function it captures only the initial snapshot—the subsequent writes to a function local go unrecorded.

The default stack VM records everything with real line numbers, so track a global there when you want the full picture.

32.6.2 `history(variable)` — View the Timeline

Once you're tracking a variable, call `history()` to retrieve its full timeline:

Listing 32.34: Viewing variable history

```
flux temperature = 20.0
track(temperature)

temperature = 22.5
temperature = 25.0
temperature = 18.3

let timeline = history(temperature)
for entry in timeline {
    print(entry)
}
```

Listing 32.35: The timeline, on the default VM

```
{"fn": nil, "line": 2, "phase": fluid, "value": 20}
{"fn": nil, "line": 4, "phase": unphased, "value": 22.5}
{"fn": nil, "line": 5, "phase": unphased, "value": 25}
{"fn": nil, "line": 6, "phase": unphased, "value": 18.3}
```

Each entry in the timeline is a `Map` with four keys:

Key	Description
`phase`	A String naming the value's phase at recording time, using the same vocabulary as `phase_of`: `"fluid"`, `"crystal"`, `"unphased"`, `"sublimated"`
`value`	A deep copy of the variable's value at that moment
`line`	The source line number where the assignment occurred (always `0` under `--tree-walk`)
`fn`	Always `nil` in v0.6.4

Two things about that output deserve a second look.

phase Is Not the Declaration Keyword—and fn Is Always nil

The phase field does not report "flux", "fix", or "let". rt_record_history() in src/runtime.c calls builtin_phase_of_str(), the same function backing phase_of, so the vocabulary is the materials one: fluid, crystal, unphased, sublimated.
Watch what that means above: only the *first* snapshot reads fluid. track takes it straight from the flux binding, phase tag and all. Every later snapshot records the phase of the freshly computed value being assigned, and a plain float literal carries no phase—so they read unphased. That is the field telling the truth about values, not about declarations.
The fn field is present in every entry and is nil in every entry, at top level and inside functions alike, on all three backends. The same line of rt_record_history() that fills in the phase sets fn_name = NULL unconditionally. Don't build a report around it.

The phases() function is a synonym for history() that returns the same data. Use whichever name feels more natural.

32.6.3 rewind(variable, n) — Look Back in Time

The rewind() function retrieves the value of a variable from n steps ago:

Listing 32.36: Rewinding variable state

```
flux score = 0
track(score)

score = 10
score = 25
score = 50
score = 100

print(rewind(score, 0))  // 100 (current)
print(rewind(score, 1))  // 50  (one step back)
print(rewind(score, 2))  // 25  (two steps back)
print(rewind(score, 3))  // 10  (three steps back)
print(rewind(score, 4))  // 0   (initial value)
print(rewind(score, 99)) // nil (past the beginning)
```

The parameter n counts backward from the most recent snapshot. rewind(score, 0) returns the current value; rewind(score, 1) returns the previous value, and so on. If n exceeds the number of recorded snapshots, rewind() returns nil. That output is identical on all three backends—rewind

walks the same snapshot array everywhere, and the backend differences described above concern which snapshots get taken, not how they're indexed.

The returned value is a *deep copy*—modifying it won't affect the original variable or the history.

32.6.4 Practical Example: Debugging a Running Total

Let's use value history to find where a running total goes wrong. Two constraints shape how we write this. Lattice has no map literals, so each transaction is built with `Map::new()` and index assignment. And because `--regvm` doesn't record writes to function locals, we keep `total` at the top level, which makes the example behave identically on all three backends.

Listing 32.37: Debugging with value history

```
fn make_tx(kind: String, amount: Float) -> Map {
    let tx = Map::new()
    tx["type"] = kind
    tx["amount"] = amount
    return tx
}

flux total = 0.0
track(total)

let transactions = [
    make_tx("credit", 100.0),
    make_tx("credit", 50.0),
    make_tx("debit", 30.0),
    make_tx("credit", -20.0)
]

for tx in transactions {
    let amount = tx.get("amount")
    if tx.get("type") == "credit" {
        total = total + amount
    } else {
        total = total - amount
    }
}

// Something's wrong with the total. Let's inspect the history:
for entry in history(total) {
    print(format("line {}: total = {} ({})",
        entry.get("line"), entry.get("value"), entry.get("phase")))
}
print(format("Final total: {}", total))
```

Listing 32.38: The timeline tells the story

```
line 9: total = 0 (fluid)
line 21: total = 100 (unphased)
line 21: total = 150 (unphased)
line 23: total = 120 (unphased)
line 21: total = 100 (unphased)
Final total: 100
```

Read that timeline top to bottom and the bug names itself. The one entry from line 23 is the debit, and it behaves: 150 down to 120. The last entry comes from line 21—the *credit* branch—and the total goes *down*, from 120 to 100. A credit of −20 is a debit wearing the wrong label, and the history pins it to a specific branch on a specific line without a single `print` inside the loop.

That is the shape of the technique: you don't need to know which iteration went wrong ahead of time, only which variable to watch.

32.6.5 Combining Tracking with the Debugger

Value history and the interactive debugger complement each other, and they complement each other in a specific way: history works where `eval` doesn't.

`eval` can't reach a function local (Section 32.1.5), and a watch expression stalls the program (Section 32.4.1). `track` has neither problem, because it isn't an evaluation at all—it's a recording the runtime makes as the program runs, which you read afterwards. For "how did this value get here?", reach for `track` first and the debugger second.

The two do compose, though. Given a script that tracks a top-level accumulator over the samples `[10, 20, 30, 40]` and prints it on line 9, we can stop there and interrogate the past:

Listing 32.39: Using track() with the CLI debugger

```
$ clat --break 9 analytics.lat
Lattice debugger attached.
Breakpoints: line 9

Breakpoint 1 at line 9
-->    9 | print(running_sum)

(dbg) eval running_sum
=> 100

(dbg) eval rewind(running_sum, 3)
=> 10
```

The current value is 100 and three steps back it was 10, without re-running anything.

Remember the cost: that `eval` succeeded, so it's harmless, but it still consumes one `c`. And keep the argument to `eval` a global—`eval history(some_local)` fails on the local, with the consequences described in Section 32.1.5.

For inspecting history at a breakpoint inside a function, `breakpoint()` under `--tree-walk` is the better tool: its prompt evaluates expressions against the live environment, locals and all.

Memory Cost of Tracking

Each tracked assignment creates a deep copy of the variable's value—`value_deep_clone()`, unconditionally, with no cap on how many snapshots accumulate. For large data structures in tight loops, this can consume significant memory: tracking a 10,000-element array through a 1,000-iteration loop keeps a thousand full copies of it alive. Use `track()` judiciously—enable it for the specific variables you need to debug, and remove it when you're done.

32.7 Diagnostics Without a Prompt: `-stats` and the GC Flags

Not every bug wants a breakpoint. Memory that grows without bound, a program that slows down as it runs, a value freed too early—these show up in aggregate, not at a line. For those, Lattice has flags that instrument a whole run and report at the end, and unlike the debugger, they work on every backend.

32.7.1 -stats

Add `--stats` to any run and the interpreter prints a memory report to *stderr* after the program finishes. Here it is on a script that builds two thousand strings into an array and freezes the result:

Listing 32.40: –stats on the default stack VM

```
$ clat --stats rows.lat

=== Memory Statistics ===

Memory footprint:
  region peak:  1
  region live:  1 (147456 bytes data)
  region total: 1 created
  RSS peak:     3760 KB

Garbage collection:
  gc cycles:    0
  swept total:  0 objects
  live objects: 0
  bytes alloc:  0
2000
```

The `2000` at the bottom is the program's own output. It appears last because the report goes to stderr and the program's `print` goes to stdout; separate them with `2>stats.txt` if the interleaving bothers you.

`region` counts crystal regions—the refcounted blocks that frozen containers materialize into. One `freeze` of one array produces exactly one, which is a useful sanity check when you're reasoning about where your crystals live. Turn regions off and the count goes to zero:

Listing 32.41: –no-regions removes the region machinery

```
$ clat --no-regions --stats rows.lat
  region peak:  0
  region live:  0 (0 bytes data)
  region total: 0 created
```

`--stats` Reports on All Three Backends

The README's flag table still describes `--stats` as tree-walk-only. That is stale; as of v0.6.4 all three backends report.

What differs is how much they say. The default stack VM prints **Memory footprint** and **Garbage collection**. `--regvm` prints **Memory footprint** alone—no GC section at all. `--tree-walk` is by far the most instrumented, adding **Phase transitions** (freezes, thaws, deep clones), **Allocations** by kind, fluid-heap peak/live/total with a **churn ratio**, **Scope lifecycle** (pushes, pops, peak depth), **Calls & bindings**, forge-block counts, and a **Timing** section breaking out GC, freeze, and thaw milliseconds.

That asymmetry is a genuine reason to reach for `--tree-walk`: when the question is "how many times am I deep-cloning this?" or "how much fluid memory am I churning?", it is the only backend that will tell you. The same script above reports `freezes: 1`, `deep clones: 0`, a churn ratio of 641.5x, and `freeze total: 3.279 ms` under `--tree-walk`.

32.7.2 The GC Flags

Five flags control garbage collection, and each one is a different diagnostic instrument.

Flag	Effect
`--gc`	Enable the collector
`--no-gc`	Disable the collector
`--gc-stress`	Collect at every allocation checkpoint instead of at a heap threshold
`--gc-incremental`	Tri-color incremental collection, bounding per-statement pauses; implies `--gc`
`--no-regions`	Disable region-based memory for crystals

`--gc-stress` is the one you'll use for correctness work. Pair it with `--stats` and you can watch it do its job—the same two-thousand-push script that reports `gc cycles: 0` on a normal run reports this under stress:

Listing 32.42: –gc-stress, measured

```
$ clat --gc-stress --stats rows.lat
Garbage collection:
  gc cycles:    2004
```

Roughly one collection per allocation. The point isn't the number; it's that a use-after-free which normally hides behind a comfortable delay before the next collection now fires on the very next allocation. If a program is correct under `--gc-stress`, it is unlikely to have a rooting bug. It is also dramatically slower, which is why this belongs in a periodic job rather than every build. `clat test --gc-stress` accepts the flag too (Section 31.5.7).

`--gc-incremental` answers a different question: not "is this correct?" but "why did this pause?" It spreads collection work across statements so no single one absorbs a full mark-and-sweep, and spawned-thread VMs inherit the configuration.

GC No Longer Fears Refs and Iterators

Older guidance warned against combining the collector with `Ref` values or iterators. As of v0.6.4 the collector traces iterators, Refs, tuples, and upvalues, and reference cycles are collected. Turn the GC on and leave it on.

Exercises

1. Write a program that computes the factorial of a number using recursion. Run it under `--debug` and step through the recursive calls using `s`. Use `bt` at the deepest recursion level to see the full call stack. Then use `o` to step out one frame at a time, watching the frame count in `bt` shrink.

2. Create a program with a loop that processes an array of numbers, keeping the loop counter as a top-level `flux` binding. Set a conditional breakpoint that only triggers when the counter reaches a specific value, and confirm with `p` that it fired on the iteration you meant. Then move the counter inside a function and observe the `Warning: breakpoint 1 condition error` that results.

3. Add `track()` to a top-level variable driven by a sorting algorithm (e.g., insertion sort). After the sort completes, print the full `history()` and use `rewind()` to inspect the variable's value at three different points. Run the same program under all three backends and note which line numbers survive.

4. Drive the DAP server by hand. Write a script in any language that spawns `clat --dap prog.lat`, sends `initialize`, `launch`, `setBreakpoints`, and `configurationDone` with Content-Length framing, and prints every message that comes back. Compare the `stopped`, `output`, and `terminated` events against the order you expected.

5. Write a program that calls `breakpoint()` inside an error-handling path. Run it under all three backends and record the three different outcomes. Then answer for yourself: what happens if this program runs in a pipeline with no interactive stdin?

6. Reproduce the watch stall. Put a breakpoint inside a five-iteration loop and continue through it, recording the counter at each stop. Add a watch on any expression and repeat. Then `unwatch` it and count how many `c` commands it takes before the counter moves again.

What's Next

You now have a complete toolkit for developing Lattice programs: formatting (`clat fmt`), documentation (`clat doc`), testing (`clat test`), and debugging (`--debug`, DAP, `track`, and `--stats`).

You have also collected a surprising number of "this depends on the backend" footnotes: `clat test` runs on the tree-walker, the debugger runs only on the stack VM, `breakpoint()` behaves three different ways, `track` records line numbers on two backends out of three, and `--stats` is chattiest where you might least expect. Those aren't arbitrary—each one falls out of how that backend represents a call frame, a local slot, or a heap. In Part X, we'll go under the hood and explore the internals of the Lattice runtime itself, at which point the footnotes start to look like consequences. Chapter 33 starts with the three compilation backends—tree-walk interpreter, stack VM, and register VM—and explains how your Lattice code is transformed from source text into executable instructions.

Part X

Under the Hood

Chapter 33

The Three Backends

Every line of Lattice code you write takes the same journey through the front end: the *lexer* turns source text into tokens, the *parser* builds an abstract syntax tree (AST), and the optional *phase checker* validates mutability constraints. What happens *after* that point—how the AST actually gets executed—depends on which **backend** you choose.

Lattice ships with three of them:

1. A **tree-walk interpreter** that traverses the AST node by node.
2. A **stack-based bytecode VM** that compiles the AST to a stream of one-byte opcodes and executes them against a value stack.
3. A **register-based VM** that compiles to fixed-width 32-bit instructions and operates on virtual registers.

Why three? Because each backend occupies a different point in the trade-off space between implementation complexity, startup latency, and peak throughput. In this chapter we will open each one up, see how it works, and learn when to reach for it.

33.1 Tree-Walk Interpreter

33.1.1 How It Works

The tree-walk interpreter is the oldest backend in Lattice. It lives in `src/eval.c` and is centred on a single recursive function, `eval_expr_inner`, that pattern-matches on the AST node type and returns an `EvalResult`. Statements are handled by `eval_stmt`; blocks by `eval_block_stmts`.

Tree-Walk Interpretation

A *tree-walk interpreter* executes a program by recursively visiting each node of the abstract syntax tree produced by the parser. There is no compilation step and no intermediate bytecode—the AST *is* the program representation at runtime.

Consider this tiny program:

Listing 33.1: A value flowing through the tree-walker

```
let width = 5
let height = 3
let area = width * height
print(area) // 15
```

When the tree-walker encounters the expression `width * height`, it:

1. Recognises a `BINARY_MUL` expression node.
2. Recursively evaluates the left child (`width`), looking it up in the current `Env` and finding the integer 5.
3. Recursively evaluates the right child (`height`), finding 3.
4. Multiplies the two values and wraps the result in an `EvalResult`.

Each recursion pushes a native C call frame. That means deeply nested expressions or long chains of function calls consume real stack space—the `Evaluator` struct tracks call depth and caps it at `max_call_depth` to prevent a segfault.

33.1.2 Environment and Scoping

The tree-walker manages variables through an `Env` (environment) structure—a hash map of names to `LatValues` with a pointer to the enclosing scope. Entering a block pushes a new child `Env`; leaving it pops the environment, freeing any bindings that went out of scope.

Listing 33.2: Scoping in the tree-walker

```
let outer = "hello"
if true {
  let inner = "world"
  print("${outer} ${inner}") // hello world
}
// inner is no longer accessible here
```

The tree-walker also tracks a rich set of metadata in its `Evaluator` struct: struct definitions (`struct_defs`), enum declarations (`enum_defs`), trait and impl registries, phase reactions, bonds, seed contracts, pressure constraints, and a full dual-heap garbage collector with region support. For a long stretch of Lattice's history, this made the tree-walker the most feature-complete backend. That gap has closed: as of v0.6.4, all three backends run the full language—concurrency, pattern matching, `defer`, the phase system, everything. What the tree-walker retains is its role as the *reference implementation*: the most direct transcription of the language's semantics into code.

33.1.3 When to Use It

Invoke the tree-walker with the `--tree-walk` flag:

Listing 33.3: Running with the tree-walk backend

```
// command line:
// clat --tree-walk my_program.lat
```

Reach for the tree-walker when you need:

- **The richest memory profile.** As of v0.6.4, all three backends accept `--stats`, but the tree-walker's report goes furthest: phase transitions (freezes, thaws, deep clones), scope push/pop counts and peak depth, per-operation timing for freeze and thaw, and GC sweep detail. The bytecode VMs report region usage, RSS peak, and GC cycle counts.
- **A second opinion.** When a program behaves unexpectedly, running it under `--tree-walk` tells you whether the behaviour comes from the language semantics or from a backend quirk. The tree-walker executes the AST directly, so its run is the most literal reading of your source.
- **Semantics-first error reporting.** Because execution follows the AST directly, the tree-walker's error messages can reference the exact AST node that failed, which is sometimes more informative than a bytecode offset.

Tree-Walker Performance

The tree-walker is the slowest backend. Every expression evaluation involves pointer chasing through AST nodes, C-level recursion overhead, and hash-map lookups for every variable access. For compute-heavy programs, the bytecode VMs are typically 3–10× faster (see Section 33.5).

33.2 Stack-Based Bytecode VM

The stack-based bytecode VM is Lattice's **default backend**. When you run `clat my_program.lat` without any flags, this is the engine that fires.

33.2.1 Compilation

Before execution, the stack compiler (`src/stackcompiler.c`) walks the AST and emits a flat stream of bytes into a `Chunk` structure (defined in `include/chunk.h`). A `Chunk` holds:

- `code` — a dynamic array of `uint8_t` values: the bytecode stream.
- `constants` — a pool of `LatValues` referenced by index from the bytecode.
- `lines` — a parallel array mapping each byte offset to a source line number (used for error messages and debugging).
- `local_names` — debug metadata mapping stack slots to variable names.
- `pic` — a polymorphic inline cache for method dispatch (more on this shortly).

The compiler maintains a `Compiler` struct that tracks local variables as stack slots, manages scope depth, resolves upvalues for closures, and handles break/continue jump patching. Each function body is compiled into its own child `Chunk`, stored as a closure constant in the parent chunk's constant pool.

Listing 33.4: A function compiled to stack bytecode

```
fn add(a: Int, b: Int) -> Int {
  return a + b
}
let result = add(10, 20)
print(result) // 30
```

This compiles roughly to:

```
; Top-level chunk:
```

```
OP_CLOSURE       0       ; Push closure for add()
OP_DEFINE_GLOBAL "add"   ; Bind it globally
OP_GET_GLOBAL    "add"   ; Push the closure
OP_CONSTANT      10      ; Push argument 1
OP_CONSTANT      20      ; Push argument 2
OP_CALL          2       ; Call with 2 arguments
OP_DEFINE_GLOBAL "result"
OP_GET_GLOBAL    "result"
OP_PRINT         1       ; Print 1 value

; Function chunk for add(a, b):
OP_GET_LOCAL     1       ; Push a (slot 1; slot 0 = function itself)
OP_GET_LOCAL     2       ; Push b
OP_ADD                   ; Pop both, push sum
OP_RETURN                ; Return TOS
```

33.2.2 Opcodes

The full opcode set is defined in `include/stackopcode.h`. Opcodes are single bytes (0–255), giving the stack VM a compact instruction stream. Here are the major families:

Table 33.1: Stack VM opcode families

Family	Examples	Purpose
Stack manipulation	`OP_CONSTANT`, `OP_POP`, `OP_DUP`	Load values, discard or duplicate the top of stack
Arithmetic/logic	`OP_ADD`, `OP_MUL`, `OP_NOT`	Pop operands, push result
Comparison	`OP_EQ`, `OP_LT`, `OP_GTEQ`	Pop two values, push boolean
Variables	`OP_GET_LOCAL`, `OP_SET_GLOBAL`	Read/write locals (by slot) and globals (by name)
Control flow	`OP_JUMP`, `OP_LOOP`, `OP_JUMP_IF_FALSE`	Conditional and unconditional jumps
Functions	`OP_CALL`, `OP_CLOSURE`, `OP_RETURN`	Call, create closures, return
Data structures	`OP_BUILD_ARRAY`, `OP_BUILD_STRUCT`	Construct compound values from stack elements
Phase system	`OP_FREEZE`, `OP_THAW`, `OP_CLONE`	Phase transitions and deep cloning
Optimised paths	`OP_ADD_INT`, `OP_INC_LOCAL`, `OP_LOAD_INT8`	Fast paths for integer arithmetic

Some opcodes use one-byte operands (e.g. `OP_GET_LOCAL` takes a slot index); some use two-byte big-endian operands for larger ranges (`OP_JUMP`, `OP_LOOP`). When the constant pool exceeds 255 entries, wide variants like `OP_CONSTANT_16` kick in with 16-bit indices.

Exploring Opcodes

For a complete listing of every opcode with its encoding and semantics, see Chapter B. You can also call the C function `opcode_name()` (from `include/stackopcode.h`) to get the human-readable name of any opcode byte.

33.2.3 The Value Stack

At the heart of the stack VM is a fixed-size array of 4,096 `LatValue` slots:

```
LatValue stack[STACKVM_STACK_MAX];   // 4096 slots
LatValue *stack_top;                  // points past the last used slot
```

Every computation goes through this stack. To add two numbers, the VM pushes both onto the stack, then the `OP_ADD` handler pops them, adds them, and pushes the result. Local variables live directly in the stack—each call frame records a `slots` pointer marking the base of its window into the value stack.

Listing 33.5: Visualising the value stack

```
fn multiply(x: Int, y: Int) -> Int {
  let product = x * y
  return product
}
let answer = multiply(6, 7)
print(answer) // 42
```

When `multiply` is executing, the stack looks roughly like this:

```
slot 0: <multiply closure>     <-- frame->slots
slot 1: 6                      (x)
slot 2: 7                      (y)
slot 3: 42                     (product)
                               <-- stack_top
```

33.2.4 Call Frames

The VM maintains up to 512 call frames (`STACKVM_FRAMES_MAX`), each a `StackCallFrame` containing:

- `chunk` — the `Chunk` being executed in this frame.
- `ip` — the instruction pointer, pointing into the chunk's bytecode.
- `slots` — the base of this frame's region on the value stack.
- `upvalues` — pointers to closed-over variables for closures.

When a function returns via `OP_RETURN`, the VM pops the frame, restores the caller's instruction pointer, and pushes the return value onto the caller's stack.

33.2.5 The Dispatch Loop

The workhorse of the stack VM is the dispatch loop in `src/stackvm.c`—function `stackvm_run`. It reads one opcode at a time and branches to the handler for that opcode.

On compilers that support it (`GCC`, `Clang`), the dispatch loop uses **computed gotos**: a table of label addresses indexed by opcode value. After each handler finishes, it jumps directly to the next handler through the table, avoiding the overhead of a `switch` statement's bounds check and indirect branch prediction penalty. This is defined under the `VM_USE_COMPUTED_GOTO` preprocessor guard.

Computed Gotos

Computed gotos (gcc's "labels as values" extension) let the VM replace a central `switch` with a dispatch table of `void*` pointers. Each handler ends with `goto *dispatch_table[next_opcode]`, which modern CPUs can predict more accurately than a single indirect branch. The result is typically a 10–20% speed improvement on tight loops.

33.2.6 Exception Handling and Defer

The stack VM implements `try`/`catch` with an exception handler stack (`STACKVM_HANDLER_MAX` = 64 entries). When the compiler encounters a `try` block, it emits `OP_PUSH_EXCEPTION_HANDLER` with a jump offset to the catch body. If `OP_THROW` fires, the VM unwinds frames back to the handler, builds a structured error map (with `message`, `line`, and `stack` fields), and pushes it onto the value stack for the catch clause.

`defer` works similarly: `OP_DEFER_PUSH` records the defer body's bytecode offset and scope depth. When execution leaves the scope (via `OP_DEFER_RUN`, return, or exception unwinding), deferred blocks execute in LIFO order, sharing the parent frame's stack slots.

Listing 33.6: Exception handling and defer in action

```
fn read_config(path: String) -> Map {
  defer { print("finished with ${path}") }

  try {
    let text = read_file(path)
    if text == nil { return Map::new() }
    return json_parse(text)
  } catch err {
    print("bad config: ${err["message"]}")
    return Map::new()
  }
}

let config = read_config("settings.json")
print(config["retries"])   // 3
let fallback = read_config("broken.json")
print(fallback.len())      // 0
```

Run against a valid `settings.json` and a truncated `broken.json`, this prints:

```
finished with settings.json
3
bad config: json_parse error at position 12: unexpected character
finished with broken.json
0
```

Note the ordering: the `catch` clause runs first, then the deferred block fires as the function's scope unwinds—the same on every backend.

33.2.7 Upvalues and Closures

When a closure captures a variable from an enclosing scope, the compiler creates an `ObjUpvalue`. While the captured variable is still on the stack (i.e. the enclosing function has not returned), the upvalue's `location` pointer points directly into the stack slot. When the enclosing scope ends, `OP_CLOSE_UPVALUE` copies the value into the upvalue's `closed` field and redirects `location` to point there. This "open → closed" transition lets closures survive their parent's return without dangling pointers.

Listing 33.7: Closures capture variables via upvalues

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

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

After `make_counter` returns, the local `count` is closed into the upvalue. Each call to the returned closure reads and writes through the same `ObjUpvalue.closed` field.

Empty Parameter Lists

An empty closure parameter list is written `| |`—with a space between the bars. The token `||` is logical OR, so `return || { ... }` is a parse error, not a closure. The return type annotation, meanwhile, uses the capital-F `Fn() -> Int` form introduced in v0.3.29.

33.3 Register-Based VM

The register-based VM is Lattice's newest backend. Invoke it with the `--regvm` flag. Where the stack VM manipulates an implicit operand stack, the register VM names its operands explicitly in each instruction—more like a physical CPU.

33.3.1 Instruction Encoding

Every register VM instruction is a fixed-width **32-bit word**, defined as `typedef uint32_t RegInstr` in `include/regopcode.h`. The encoding packs an 8-bit opcode and up to three operand fields:

Encoding and decoding use bit-manipulation macros:

```
// Encode: ROP_ADD into register 3, from registers 1 and 2
REG_ENCODE_ABC(ROP_ADD, 3, 1, 2)
//  => 0x02010308  (C=2, B=1, A=3, op=ROP_ADD)
```

Table 33.2: Register VM instruction formats

Format	Layout (bits)	Use
ABC	`[op:8][A:8][B:8][C:8]`	Three-address ops: `R[A] = R[B] + R[C]`
ABx	`[op:8][A:8][Bx:16]`	Load constant: `R[A] = K[Bx]`
AsBx	`[op:8][A:8][sBx:16]`	Conditional jump: `if !R[A]: ip += sBx`
sBx	`[op:8][sBx:24]`	Unconditional jump

```
// Decode:
REG_GET_OP(instr)   // opcode byte
REG_GET_A(instr)    // destination register
REG_GET_B(instr)    // first source
REG_GET_C(instr)    // second source
```

Fixed-width instructions mean the VM can index into the code array by instruction number rather than scanning variable-length byte sequences. This gives better instruction-cache behaviour and eliminates the need for "wide" opcode variants.

33.3.2 Register Windows

Each call frame in the register VM gets its own **register window**—a contiguous slice of 256 `LatValue` slots carved from the global register stack:

```
LatValue reg_stack[REGVM_REG_MAX * REGVM_FRAMES_MAX];
// = 256 * 64 = 16,384 register slots
```

When a function is called, the VM advances `reg_stack_top` by `REGVM_REG_MAX` (256) and points the new frame's `regs` pointer there. Function arguments are placed into the callee's low registers by the caller; the return value is written back to a register in the *caller's* window designated by `caller_result_reg`.

Listing 33.8: Register allocation for a function call

```
fn distance(x1: Float, y1: Float, x2: Float, y2: Float) -> Float {
  let dx = x2 - x1
  let dy = y2 - y1
  return (dx * dx + dy * dy).sqrt()
}

let d = distance(0.0, 0.0, 3.0, 4.0)
print(d) // 5.0
```

Inside `distance`, the register compiler assigns:

```
R[0] = <distance closure>
R[1] = x1 (0.0)
R[2] = y1 (0.0)
R[3] = x2 (3.0)
R[4] = y2 (4.0)
R[5] = dx  (3.0, result of ROP_SUB R[5], R[3], R[1])
R[6] = dy  (4.0, result of ROP_SUB R[6], R[4], R[2])
R[7] = dx*dx (9.0)
R[8] = dy*dy (16.0)
R[9] = sum   (25.0)
```

Notice the three-address style: `ROP_SUB R[5], R[3], R[1]` computes the subtraction and writes the result directly to R[5] without touching a stack. Intermediate values stay in registers instead of being pushed and popped, reducing memory traffic.

33.3.3 The Register Compiler

The register compiler (`src/regcompiler.c`) maintains a `RegCompiler` struct that is structurally similar to the stack compiler, but with a critical addition: a **register allocator**. Two functions manage the register file:

- `alloc_reg()` returns the next available register and advances the `next_reg` pointer (also updating a high-water mark `max_reg` used for bounded initialisation).
- `free_reg()` releases a register when its value is no longer needed (temporary expression results).

Local variables are assigned a permanent register for their entire lifetime within a scope. Temporary values get registers that are freed as soon as the value is consumed. This means the compiler must decide *at compile time* where every value lives—a more complex task than the stack compiler, which only ever pushes and pops.

33.3.4 Inline Caches

Both VMs use **polymorphic inline caches** (PICs) to speed up method dispatch, but they are especially important for the register VM. The PIC system is defined in `include/inline_cache.h`.

When the VM encounters a method call like `arr.push(42)`, it must determine the receiver's type, hash the method name, and search a dispatch table. A PIC short-circuits this:

1. On the first call, the full lookup runs and the result is cached in a 4-entry `PICSlot` keyed by `(type_tag, method_hash)`.

2. On subsequent calls, the VM checks the cache first. A hit returns the `handler_id` directly—an integer that indexes into a jump table, skipping the hash and string comparison entirely.
3. If the cache has four entries and a new type appears (megamorphic site), the oldest entry is evicted in FIFO order.

Listing 33.9: A polymorphic call site

```
fn describe(items: Array) -> String {
  // items could contain mixed types
  flux parts = []
  for item in items {
    // item.to_string() is polymorphic: Int, Float, String, etc.
    parts.push(item.to_string())
  }
  return parts.join(", ")
}

print(describe([1, 2.5, "three"])) // 1, 2.5, three
```

The PIC table uses a direct-mapped scheme with 64 slots per chunk, indexed by `ip_offset & 0x3F`. Collisions between different call sites in the same chunk are benign—they cause cache misses that trigger the slow path, and nothing worse.

Handler IDs

Handler IDs are small integers defined as `#define` constants in `include/inline_cache.h`: `PIC_ARRAY_PUSH = 2`, `PIC_STRING_SPLIT = 32`, and so on. The special value `PIC_NOT_BUILTIN = 255` tells the VM to fall through to struct, map, or impl-block lookup.

33.3.5 RegChunk: The Register VM's Code Container

Where the stack VM stores bytecode in a `Chunk` of `uint8_t`, the register VM uses a `RegChunk` containing `uint32_t` instructions:

- `code` — array of `RegInstr` (32-bit words).
- `constants` — the same constant-pool design as the stack VM's `Chunk`.
- `magic` — the four bytes `0x524C4154` ("RLAT"), used to distinguish register chunks from stack chunks during serialisation.
- `max_reg` — the high-water register count, so the VM only needs to initialise that many registers per frame instead of all 256.

- `pic` — a `PICTable` for inline caching, same as the stack VM's `Chunk`.

Constant deduplication runs at compile time: the compiler scans the existing pool for matching strings, integers, and floats before adding a new entry. This keeps the pool compact and reduces memory pressure.

33.4 Choosing a Backend

33.4.1 Command-Line Flags

Table 33.3: Backend selection flags

Flag	Backend	When to use
(none)	Stack VM	General-purpose (default)
`--tree-walk`	Tree-walk interpreter	Memory profiling, full feature coverage
`--regvm`	Register VM	Performance-sensitive code

Table 33.4: Crystal sharing and memory controls

Flag / Variable	Effect
`--no-regions`	Disable crystal sharing; every freeze is a plain tag flip
`LATTICE_SHARE_CRYSTALS=0`	Env-var kill switch, same as `-no-regions`
`LATTICE_FORCE_COPY=1`	Debug oracle: every share becomes a deep copy

Listing 33.10: Selecting a backend from the command line

```
// Stack VM (default):
// clat my_program.lat

// Tree-walk interpreter:
// clat --tree-walk my_program.lat

// Register VM:
// clat --regvm my_program.lat
```

33.4.2 Feature Support

Not every feature is available on every backend. Here is a summary:

Table 33.5: Feature availability by backend

Feature	Tree-Walk	Stack VM	Reg VM
Core language (variables, functions, closures)	✓	✓	✓
Structs, enums, traits, impls	✓	✓	✓
Phase system (freeze, thaw, clone)	✓	✓	✓
Concurrency (scope/spawn, channels)	✓	✓	✓
Try/catch, defer	✓	✓	✓
Interactive debugger (`--debug`)	—	✓	—
DAP protocol (`--dap`)	—	✓	—
Memory stats (`--stats`)†	✓	✓	✓
GC modes (`--gc`, `--gc-stress`)	✓	✓	—
Pre-compiled bytecode (`.latc`/`.rlat`)	—	✓	✓

† All three backends accept `--stats` and all three print a `=== Memory Statistics ===` block, but the reports differ in depth. The default stack VM prints memory footprint (region peak/live/total, RSS peak) plus a garbage-collection summary; `--regvm` prints the memory footprint alone; `--tree-walk` adds phase transitions, allocation counts, fluid churn ratio, scope lifecycle, call and binding counts, and per-operation timing for freeze and thaw. See the list at the start of Section 33.1 and the worked example in Section 32.7. If you have read that `--stats` is tree-walk-only, that description is out of date.

33.4.3 Divergences That Remain in v0.6.4

Feature parity is not the same thing as byte-for-byte agreement. All three backends run the whole language, and they still disagree in a handful of places—sometimes only in how a refusal is worded, sometimes in whether the program runs at all. Earlier chapters flag these where they come up; here they are in one place, each one observed on the v0.6.4 binary.

Table 33.6: Behaviour that still differs between backends in v0.6.4.

Behaviour	Default (stack VM)	`--tree-walk`	`--regvm`
Runtime error prefix	`vm error:`	`error:`	`regvm error:`
Method not found, e.g. `5.push(3)`	`type 'Int' has no method 'push'`	`.push() is not defined on non-array`	`no method 'push' on Int`
Missing map key: `typeof(m[k])`	`"Nil"`	`"Unit"`	`"Nil"`

Behaviour	Default (stack VM)	`--tree-walk`	`--regvm`
`.each` and `.flatten` on arrays	Work	`unknown method '.each()' on Array`	Work
`Map.delete(key)`	`type 'Map' has no method 'delete'`	`unknown method '.delete()' on Map`	Removes the key in place
`.reduce` arity	Closure and seed both required	Both required; one argument gives `.reduce() expects 2 arguments`	Also accepts the closure alone, seeding from the first element
Callback error inside `.filter` or `.reduce`	Process aborts; `catch` never runs	Catchable, unwinds normally	Process aborts; `catch` never runs
Nested struct-field write	`inv.items.push(3)` does not reach `inv`	Reaches `inv`	Does not reach `inv`
`+=` on a `fix` local	`cannot modify a frozen value`	Succeeds	`cannot modify a frozen value`
Bare function name as a value, `let op = add`	Works	`undefined variable 'add'`	Works
`#mode strict` consume-on-freeze	Source stays readable	Source is consumed: `undefined variable 'data'`	Source stays readable
Phase-constraint refusal on a `~Map` parameter	`phase constraint violation in function 'mutate'`	`parameter 'data' requires flux argument, got crystal (fix)`	`phase constraint violation in function 'mutate'`
`breakpoint()`	Prompt opens, but a bare local name reports `undefined variable` and ends the run	Prompt opens and evaluates locals against the paused frame	`breakpoint(): not supported under the register VM backend; the program continues`
`--stats` report	Memory footprint plus GC summary	Adds phase transitions, allocations, churn, scope lifecycle, timings	Memory footprint only

Portability Rule of Thumb

Write to the intersection. Pass `.reduce` both arguments, keep mutation out of nested struct fields, use `typeof(m[k]) == "Nil"` only when you have pinned a backend, and never let a callback inside `.filter` or `.reduce` raise. Code that stays inside those lines behaves the same everywhere, which is what makes the "run it under `--tree-walk` for a second opinion" habit worth having.

33.4.4 The Execution Pipeline

Regardless of the backend you choose, the front end is shared. The pipeline in `src/main.c` looks like this:

1. **Lex**: `lexer_tokenize()` produces a vector of tokens.
2. **Parse**: `parser_parse()` produces a `Program` (the AST).
3. **Phase check** (strict mode): `phase_check()` validates phase annotations.
4. **Match check**: `check_match_exhaustiveness()` warns about non-exhaustive matches.
5. **Execute**:
   - Tree-walk: `evaluator_run(ev, &prog)`
   - Stack VM: `stack_compile(&prog, ...)` then `stackvm_run(&vm, chunk, ...)`
   - Register VM: `reg_compile(&prog, ...)` then `regvm_run(&vm, chunk, ...)`

Pre-Compiled Bytecode

You can skip steps 1–4 entirely by pre-compiling to `.latc` or `.rlat` files. We cover this in Chapter 34.

33.5 Performance Characteristics

33.5.1 Where the Time Goes

The three backends differ fundamentally in how they spend CPU cycles:

Tree-walker: dominated by pointer chasing (AST node traversal), hash-map lookups for variable access, and C-level recursion overhead. Each expression evaluation requires multiple function calls and dynamic dispatch on node types.

Stack VM: dominated by the dispatch loop's opcode fetch-decode-execute cycle. With computed gotos, the branch predictor handles the dispatch well, but every operation still requires pushing to and popping from the value stack—each a memory write and read.

Register VM: dominated by instruction decoding (bit shifts and masks) and direct register access. Because operands are named in the instruction itself, there is no push/pop overhead. The fixed-width encoding also means the instruction pointer advances by a predictable amount, improving prefetch behaviour.

33.5.2 Microbenchmark: Fibonacci

A classic (if imperfect) benchmark is recursive Fibonacci:

Listing 33.11: Fibonacci benchmark

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

print(fib(35))
```

On a typical machine, the relative performance looks roughly like this:

Table 33.7: Relative Fibonacci performance (lower is better)

Backend	Relative Time
Register VM	1.0×
Stack VM	1.2–1.5×
Tree-walk interpreter	5–20×

The register VM wins because it avoids the stack VM's push/pop traffic and because its three-address instructions express the computation more densely. The tree-walker falls far behind due to per-node recursion and hash-map variable lookups.

Benchmarks Are Not the Whole Story

Fibonacci is function-call heavy and computation heavy, which exaggerates the dispatch overhead difference. For I/O-bound programs (web servers, file processing), the backend difference may be negligible because the bottleneck is the operating system, not the VM.

33.5.3 Startup Time

The tree-walker has the lowest startup time: it runs immediately from the AST with no compilation step. The stack VM and register VM both pay a compilation cost before the first instruction executes. For small scripts (a few dozen lines), this compilation overhead can exceed the total execution time.

If startup latency matters—say, for a CLI tool that runs and exits quickly—consider pre-compiling to bytecode (see Chapter 34).

33.5.4 Memory Usage

Tree-walker: keeps the full AST in memory throughout execution, plus the environment chain (hash maps). Memory usage scales with program size and scope depth.

Stack VM: the AST is freed after compilation; only the bytecode chunks and the value stack remain. The fixed-size value stack (4,096 slots) puts a ceiling on per-frame memory, though deep recursion can exhaust it.

Register VM: uses a fixed register stack of 16,384 slots (256 registers × 64 frames). The 32-bit instructions use 4× more bytes per instruction than the stack VM's byte stream, but there are typically *fewer* instructions because three-address ops replace sequences of push/pop operations.

33.5.5 Which One Should You Choose?

For most users, the answer is: **use the default** (stack VM). It has the best balance of features, performance, and stability. It supports the interactive debugger, pre-compiled bytecode, garbage collection, and has been the most thoroughly tested backend since Lattice v0.2.

Choose `--regvm` when you have a compute-heavy workload and want maximum throughput.

Choose `--tree-walk` when you want the deepest memory instrumentation—all three backends honour `--stats`, but only the tree-walker adds phase-transition counts, allocation breakdowns, churn ratio, scope lifecycle and freeze/thaw timing (Section 32.7)—or when you need `breakpoint()` to evaluate a bare local name against a paused frame, which the default VM refuses and the register VM declines outright (Section 32.4.2). "Features the bytecode backends have not caught up on" is no longer a reason: as of v0.6.4 all three run the whole language, and what remains are the wording and edge-case differences catalogued in Section 33.4.3.

33.6 Exercises

1. Write a Lattice program that computes the sum of integers from 1 to 1,000,000 using a `for` loop. Run it with all three backends and compare the execution times using your operating system's time utility (e.g. `time clat --regvm sum.lat`).

2. Modify the Fibonacci benchmark to use memoisation (store previously computed values in a `Map`). How does memoisation affect the relative performance of the three backends? Why?

3. Using the `--tree-walk --stats` flags, profile a program that creates and freezes several structs. Examine the output to understand how many freezes, deep clones, and scope pushes/pops occurred.

4. Read the opcode listing in Chapter B. Pick any "optimised" stack VM opcode (like `OP_ADD_INT` or `OP_INC_LOCAL`) and explain why a specialised opcode for integers is faster than the generic `OP_ADD`.

5. (Challenge) The register VM's `RegChunk` stores a `max_reg` field. Why is this useful? What would happen if the VM always initialised all 256 registers per frame?

What's Next Now that we know how Lattice code gets compiled and executed, the natural next question is: can we *save* that compiled form to disk? In Chapter 34, we will explore the `.latc` and `.rlat` file formats, learn how to pre-compile programs for faster startup, and meet Lattice's self-hosted compiler.

Chapter 34

Bytecode Serialization

Every time you run `clat my_program.lat`, Lattice lexes, parses, and compiles your source code before executing it. For a small script that is perfectly fine—the compiler is fast. But for larger projects, or for deployment scenarios where you want instant startup, you can skip all of that by *pre-compiling* your program to a bytecode file and running the result directly.

That was not always possible for *every* program. Until recently, three constructs—`match`, `select`, and `defer`—compiled to opcodes that pointed back into the syntax tree, and a pointer into a tree is not something you can write to a file. The v3 format closed that hole, and closing it is the reason this chapter can talk about pre-compilation as a general capability rather than a partial one.

In this chapter we explore the `.latc` and `.rlat` file formats and how they grew from v1 to v3, walk through the serialization process byte by byte, see what the loader checks before it will execute an untrusted file, learn how to use the `clat compile` command, and meet the self-hosted compiler—a Lattice program that compiles Lattice into bytecode, and that has compiled itself into a fixed point.

34.1 The .latc and .rlat File Formats

Lattice defines two binary file formats, one for each bytecode VM:

- `.latc` — compiled stack VM bytecode. Magic bytes: `LATC` (`0x4C415443`).
- `.rlat` — compiled register VM bytecode. Magic bytes: `RLAT` (`0x524C4154`).

Both formats share the same overall structure. The implementation lives in `src/latc.c` and the header in `include/latc.h`.

34.1.1 Header

Every bytecode file starts with an 8-byte header:

Table 34.1: Bytecode file header

Offset	Size	Field	Description
0	4 bytes	Magic	`"LATC"` or `"RLAT"`
4	2 bytes	Format version	`3` for both formats in v0.6.4 (little-endian `uint16`)
6	2 bytes	Reserved	Set to 0; available for future use

Dump the first sixteen bytes of a freshly compiled pair and the header is right there in the open:

```
$ xxd -l 16 greet.latc
00000000: 4c41 5443 0300 0000 1300 0000 2a00 0020  LATC........*..

$ xxd -l 16 greet.rlat
00000000: 524c 4154 0300 0000 0b00 0000 2500 0000  RLAT........%...
```

Magic, then `03 00` for the format version, then `00 00` reserved. The `uint32` that follows is the code length—`0x13` (19) bytes of stack bytecode versus `0x0b` (11) register instructions for the same program.

The deserializer checks the magic bytes first. If they do not match, it rejects the file immediately. Then it checks the version against a *range*: anything from `LATC_MIN_FORMAT` (2) through `LATC_FORMAT` (3) is accepted, and anything outside is refused by number.

```
$ clat v1.latc
error: unsupported .latc format version: 1
```

Bytecode File Formats

A `.latc` file contains serialized stack VM bytecodes (variable-width `uint8_t` opcodes). A `.rlat` file contains serialized register VM bytecodes (fixed-width `uint32_t` instructions). Both formats embed the full constant pool, line number tables, debug information, function call metadata, parameter phase constraints, and nested function chunks.

34.1.2 Three Format Generations

The version number in that header has a story behind it, and the story is really the story of how much of Lattice could be expressed as pure bytecode.

v1 The original format. Chunks carried code, lines, constants, local names, and a chunk name. Enough for straight-line code and functions.

v2 Added function call metadata: a `fn_has_variadic` flag and const-folded default parameter values, serialized right after the chunk name. Without them a deserialized function could not tell a missing argument from a defaulted one, so impl methods with defaults or variadics did not survive a round trip.

v3 The current format, developed during the v0.6.3 window and first shipped in a tagged release with v0.6.4. Two things happened here. Chunks gained a parameter phase-constraint table, so a function declared `fn mutate(data: ~Map)` still enforces its phase contract after being loaded from disk. And—the larger change—`match`, `select`, and `defer` stopped depending on the AST. `LATC_FORMAT` and `RLATC_FORMAT` both went from 2 to 3, with `MIN_FORMAT` held at 2 so existing v2 files kept loading.

That last point deserves its own paragraph, because it is what made this whole chapter possible.

The AST-Free Milestone

For a long time three constructs could not be serialized at all. `match`, `select`, and `defer` still reached back into the syntax tree at runtime, and a reference into a tree that no longer exists once the program is on disk is worth nothing. A program that used any of them could not be pre-compiled at all.

The v3 work gave each of them dedicated opcodes on both VMs. The stack VM gained `OP_MATCH_TYPE`, `OP_MATCH_RANGE`, `OP_MATCH_FLUID`, and `OP_HAS_FIELD` for pattern tests; `OP_SELECT_CHOOSE` for arm selection; `OP_DEFER_PUSH` for deferred bodies; plus `OP_CLOSE_UPVALUE_PRESERVE` and a missing-argument marker to keep binding semantics intact. The register VM gained its mirrors: `ROP_MATCH_TYPE`, `ROP_MATCH_FIELD`, `ROP_MATCH_PHASE`, `ROP_CHECK_ENUM_VARIANT`, `ROP_SELECT_CHOOSE`, `ROP_DEFER_PUSH`.

Bodies that need their own scope are stored as *pre-compiled sub-chunks* in the constant pool, addressed by index. `OP_SCOPE` carries a spawn count, a sync index, and a list of sub-chunk indices; `OP_SELECT` carries an arm count followed by a four-byte record per arm (flags, channel index, body index, binding index). Those indices point at closure constants the serializer already knew how to write recursively—which is exactly why the format needed no new machinery to absorb the change.

With the last AST dependency gone, the *entire* language serializes.

Let's prove it on a program that uses all three of the formerly unserializable constructs at once:

Listing 34.1: full.lat — enum-payload match, defer, and select in one program

```
enum Shape { Circle(Float), Rect(Float, Float), Point }

fn area(s: Shape) -> Float {
  defer { print("measured") }
  return match s {
    Shape::Circle(r) => 3.14159 * r * r,
    Shape::Rect(w, h) => w * h,
    Shape::Point => 0.0,
    _ => -1.0
  }
}

fn main() {
  print(area(Shape::Circle(2.0)))
  print(area(Shape::Rect(3.0, 4.0)))
  print(area(Shape::Point))

  let ch = Channel::new()
  ch.send(42)
  let msg = select {
    v from ch => { "got ${v}" }
    default => { "nothing waiting" }
  }
  print(msg)
}
```

```
$ clat compile full.lat && clat full.latc
measured
12.5664
measured
12
measured
0
got 42

$ clat compile --regvm full.lat && clat full.rlat
measured
12.5664
measured
12
measured
0
got 42
```

Byte-for-byte the two files are nothing alike, and both reproduce the source run exactly—down to the ordering of `defer` output, which fires as each `area()` call unwinds and therefore lands *before* the `print` that consumes its result.

The other half of v3 is equally checkable. A phase constraint written into a signature has to keep rejecting bad arguments after the function has been through a file:

Listing 34.2: phase.lat — a constraint that must survive the round trip

```
fn mutate(data: ~Map) {
  data["k"] = 1
}

fix frozen = Map::new()
mutate(frozen)
```

```
$ clat phase.lat
vm error: phase constraint violation in function 'mutate'

$ clat compile phase.lat && clat phase.latc
vm error: phase constraint violation in function 'mutate'

$ clat compile --regvm phase.lat && clat phase.rlat
regvm error: phase constraint violation in function 'mutate'
```

Same refusal from source and from both bytecode formats—which is what that four-byte phase count at the end of every chunk buys.

Because `LATC_MIN_FORMAT` is 2, a v2 file still loads on a v0.6.4 runtime. Only v1 files were left behind.

34.1.3 Chunk Serialization (Stack VM)

After the header, the body of a `.latc` file is a serialized `Chunk`. The serializer (`serialize_chunk` in `src/latc.c`) writes these sections in order:

1. **Bytecode stream**: A `uint32_t` count followed by that many raw bytes of opcodes.
2. **Line number table**: A `uint32_t` count followed by that many `uint32_t` line numbers (one per bytecode byte, parallel to the code array).
3. **Constant pool**: A `uint32_t` count followed by tagged constants. Each constant starts with a type tag byte:
    - `0` (Int): followed by a little-endian `int64_t` (8 bytes)
    - `1` (Float): followed by a little-endian `double` (8 bytes)
    - `2` (Bool): followed by a single byte (0 or 1)
    - `3` (String): followed by a `uint32_t` length, then that many raw bytes
    - `4` (Nil): no payload
    - `5` (Unit): no payload
    - `6` (Closure): a nested function or pre-compiled sub-chunk—followed by `uint32_t` param count, a variadic flag byte, then a recursively serialized child chunk
    - `7` (Array): a v3 metadata array, used for struct field-name and field-phase tables. The element surface is deliberately scalar—no executable or nested constants belong in these tables.

- 8 (Enum): a nullary enum variant such as `Color::Red`, stored as two length-prefixed strings (enum name, variant name). Only the register compiler pools these.

4. **Local names**: A `uint32_t` count, then for each slot: a presence byte (0 or 1), and if present, a length-prefixed string.

5. **Chunk name**: A presence byte, then (if present) a length-prefixed string for the function name used in stack traces.

6. **Function call metadata (v2)**: the `fn_has_variadic` flag as one byte, then a `uint32_t` default-value count, then that many tagged constants. Defaults are const-folded scalars, so the scalar tags above are the whole vocabulary here.

7. **Parameter phase constraints (v3)**: a `uint32_t` count, then that many raw phase bytes. A count of zero means an unconstrained function, which keeps the common case down to four bytes.

Note where the tag list ends: there is no tag for a map, a struct instance, a channel, or a Ref. Constant pools hold literals and code, and nothing that has to be alive to mean anything.

Listing 34.3: A program whose bytecode we will examine

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

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

When this program is compiled, the top-level chunk contains the bytecodes for the script body (defining and calling `greet`). Run `strings` over the resulting 489-byte file and the constant pools give themselves up:

```
$ strings -t d greet.latc
      0 LATC
    268 String
    279 function 'greet' parameter 'name' expects type String, got %s
    345 Hello,
    363 function 'greet' return type expects String, got %s
    424 name
    439 greet
    458 greet
    468 Lattice
```

There is the interpolation fragment `"Hello, "`, the type name `"String"`, the argument `"Lattice"`, the local slot name `"name"`, and the chunk name `"greet"` twice—once as a constant and once as the child chunk's own name field, which is what stack traces read. The two `%s` templates are the runtime type-check messages the compiler baked in for the parameter and return annotations. And wrapped around most of that, invisible to `strings`, is the `TAG_CLOSURE` entry that recursively contains the function body's chunk.

Recursive Chunk Embedding

Function bodies are stored as closure constants in their parent chunk's constant pool. When the serializer encounters a `TAG_CLOSURE`, it recursively writes the child chunk. The deserializer mirrors this: it recursively reads the child chunk and constructs a `VAL_CLOSURE` value in the parent's constant pool. This tree of chunks mirrors the lexical nesting of functions in your source code.

34.1.4 RegChunk Serialization (Register VM)

The `.rlat` format follows the same pattern but with a few differences:

- **Instructions** are serialized as `uint32_t` values (4 bytes each, little-endian) instead of single bytes. The line table is one entry per *instruction* rather than one per byte.
- **Closure constants** include an additional `uint32_t` upvalue count.
- **max_reg**: A `uint8_t` recording the high-water register count, written after the local names. This lets the VM initialize only the registers that are actually used, rather than clearing all 256 slots per frame.
- **Enum constants**: the register compiler pools nullary variants such as `Color::Red` as pre-built values, which is why the `8` tag exists at all.

- **v3 tail**: a default-parameter *count* (register-VM default expressions are bytecode, so only the arity matters), the parameter phase table, and finally the optional chunk name.

```
$ clat compile --regvm greet.lat     # produces greet.rlat
$ clat greet.rlat
Hello, Lattice!
```

The `.rlat` File Is Not the Bigger One

It is natural to assume that four-byte instructions produce a fatter file than one-byte opcodes. They do not, because the register VM needs far fewer instructions and stores far fewer line entries:

```
$ ls -l three.latc three.rlat
-rw-r--r--  1324  three.latc
-rw-r--r--  1275  three.rlat
```

For a three-function program the register file came out roughly four percent *smaller*. The stack format pays a `uint32_t` line entry for every single bytecode byte; the register format pays one per instruction. That difference swamps the instruction-width difference on ordinary code.

34.1.5 Endianness and Portability

Integers in both formats are stored in **little-endian** byte order, written out one byte at a time (`v & 0xff`, `(v >> 8) & 0xff`, and so on) rather than by copying platform memory. Lengths, counts, constant indices, and `int64_t` literals are therefore identical regardless of the machine that produced them.

Floating-point constants are the one exception worth knowing about: `bb_write_f64_le` copies the eight bytes of the `double` as the platform represents them. On every architecture Lattice targets that is IEEE-754 little-endian, so a bytecode file compiled on an ARM Mac loads and runs on an x86 Linux machine. A hypothetical big-endian port would need to byte-swap there. (The self-hosted compiler, interestingly, does not have this asterisk—it converts a float to its bit pattern with `float_to_bits` and writes the result through its explicit little-endian integer path.)

Inspecting Bytecode Files

You can use a hex editor or `xxd` to inspect the raw bytes of a `.latc` file. Look for the `LATC` magic at the start, then the `uint32_t` code length to know where the bytecodes end and the line table begins. `strings -t d` is the quicker tool when you only want to see which literals a program carries.

34.2 Wide Constant Opcodes

A constant-pool index that fits in one byte tops out at 255, and real programs blow past that regularly. The stack VM's answer is a set of paired opcodes: the narrow form takes a one-byte index, the wide form takes a two-byte big-endian index, and the compiler picks whichever fits.

Table 34.2: Wide constant opcodes in the stack VM

Opcode	Value	Purpose
`OP_CONSTANT_16`	93	Push `constants[BE16]`
`OP_GET_GLOBAL_16`	94	Read a global by name constant
`OP_SET_GLOBAL_16`	95	Write a global by name constant
`OP_DEFINE_GLOBAL_16`	96	Define a global by name constant
`OP_CLOSURE_16`	97	Build a closure from a function constant

The narrow and wide members of each pair are interchangeable to the deserializer, which is why a program with four hundred string literals compiles, serializes, and runs without ceremony:

```
$ clat compile wide.lat && clat wide.latc
constant number 0 / constant number 399
```

Not Every Operand Got a Wide Form

Field, method, and name operands—the ones behind `OP_GET_FIELD`, `OP_IMPORT`, and friends—are still a single byte. When a chunk accumulates more than 255 constants *before* one of those instructions, the compiler stops instead of truncating the index:

```
compile error: too many constants in one chunk for a
field/method/name operand (>255)
```

The register VM refuses the same program with advice attached: `too many constants in one chunk for an 8-bit name operand (split the function or reduce literals)`. The tree-walking interpreter, which has no constant pool at all, runs it happily—so this is a limit you can hit only by pre-compiling, and one that splitting a giant function immediately resolves.

34.3 Pre-Compiling for Faster Startup

34.3.1 The Compile Command

The `clat compile` subcommand takes a `.lat` source file and writes a bytecode file:

```
# Compile for the stack VM (default) -> my_program.latc
clat compile my_program.lat

# Compile with a custom output path
clat compile my_program.lat -o build/app.latc

# Compile for the register VM -> my_program.rlat
clat compile --regvm my_program.lat
```

The command carries its own documentation:

```
$ clat compile --help
Usage: clat compile [options] <file.lat> [-o output]

Compile Lattice source to bytecode.

Options:
  --regvm           Compile for register VM (output: .rlat)
  -o <path>         Set output file path
  -h, --help        Show this help

By default, compiles for the stack VM (output: .latc).
```

Under the hood, `clat compile` follows the standard pipeline—lex, parse, compile—but instead of executing the bytecode, it serializes the resulting chunk to a file. The implementation lives in `src/main.c`: the compile subcommand handler reads the source, tokenizes and parses it, compiles to a `Chunk` (or `RegChunk` with `--regvm`), and calls `chunk_save()` or `regchunk_save()` to write the binary.

34.3.2 Running Pre-Compiled Bytecode

Once you have a bytecode file, you can run it directly:

```
$ clat greet.latc
Hello, Lattice!

$ clat greet.rlat
Hello, Lattice!
```

When Lattice sees a `.latc` or `.rlat` extension it skips lexing, parsing, and compilation entirely. It reads the first four bytes, dispatches on the magic—`LATC` to the stack VM, `RLAT` to the register VM—deserializes, and begins execution.

The File Picks the Backend, Not the Flag

Because the magic number selects the VM, `--regvm` and `--tree-walk` have nothing left to decide and are ignored for pre-compiled files. All four of these do exactly the same thing:

```
clat greet.rlat
clat --regvm greet.rlat
clat --tree-walk greet.rlat     # still the register VM
clat --tree-walk greet.latc     # still the stack VM
```

You choose your backend at compile time, in the shape of the artifact you ship. There is no tree-walking bytecode format—the tree-walker executes an AST, which is precisely the thing these files exist to avoid carrying.

Here is a build script that pre-compiles every source file in a directory. It uses `exec_argv`, which takes a program and an argument array rather than a shell string, and returns a Map:

Listing 34.4: build.lat — pre-compile a project

```
import "fs" as fs

let sources = fs.glob("src/*.lat")
for file in sources {
  let output = file.replace(".lat", ".latc")
  print("compiling ${file} -> ${output}")
  let r = exec_argv("clat", ["compile", file, "-o", output], nil)
  if r["exit_code"] != 0 {
    eprint("failed: ${r["stderr"]}")
  }
}

// compiling src/greet.lat -> src/greet.latc
// compiling src/three.lat -> src/three.latc
```

Two details in that listing are load-bearing. The result is a Map, so its fields come out through index syntax—`r["exit_code"]`, not `r.exit_code`. And a non-zero exit code is *data*, not a raised error: `exec_argv` hands you the child's status and lets you decide what it means.

Bytecode Resolves Imports Against Your Working Directory

Pre-compiling does not freeze a program's import paths. A `.latc` file still resolves each `import` when it reaches the instruction, and it resolves it the same way the stack VM always does—against the process working directory (Section 28.5.2).

```
$ cd mods && clat main.latc
5
6
3.14159

$ cd .. && clat mods/main.latc
vm error: import: cannot find 'math_utils.lat'
```

Ship a multi-module program as bytecode and you are shipping a working-directory requirement along with it. Bundle a launcher that `cd`s to the install root, or use absolute import paths.

Version Compatibility

Pre-compiled files are tied to a format version, not to a Lattice release. The loader accepts anything in the range `LATC_MIN_FORMAT` to `LATC_FORMAT`, which today means v2 and v3 both load and v1 does not:

```
error: unsupported .latc format version: 1
```

The register format's equivalent is blunter—`error: unsupported .rlatc format version`, with no number. When you upgrade Lattice, recompiling is the cheap and certain move.

34.3.3 When to Pre-Compile

Pre-compilation is most useful when:

- **Deployment**: You are shipping a Lattice program to a server or container and want the fastest possible startup. The bytecode file is all you need—no source code required.
- **CI/CD pipelines**: Pre-compile once, run many times in different environments.
- **Large programs**: For projects with thousands of lines, skipping the parse and compile steps saves noticeable time.
- **Protecting source**: Distributing `.latc` files means not distributing `.lat` files. Set your expectations accordingly, though—the format is neither encrypted nor obfuscated, and it deliberately keeps every function name, local variable name, string literal, and line number so that stack

traces stay useful. The `strings` dump earlier in this chapter gave up a nine-line program's entire vocabulary in one command.

For development, pre-compilation is usually unnecessary—the compiler is fast enough that the overhead is imperceptible for most programs.

34.3.4 The Thin Runtime

If all you need to do is *run* bytecode, there is a smaller binary for the job. `clat-run` executes `.latc` and `.rlat` files, sniffing the magic exactly as `clat` does, and contains no lexer, parser, or compiler at all:

```
$ clat-run greet.latc
Hello, Lattice!

$ clat-run greet.rlat
Hello, Lattice!

$ clat-run greet.lat
error: expected a .latc or .rlat bytecode file, got 'greet.lat'
```

It is a separate build target rather than something you get from a plain `make`, and it is meaningfully smaller than the full interpreter—on the v0.6.4 release artifacts, roughly 670 KB on macOS and 750 KB on FreeBSD against a 1.2 MB `clat`. For a container image that ships one pre-compiled program, that is the whole toolchain you need.

34.4 Load-Time Verification

A bytecode file is a file. Files get truncated by a full disk, corrupted by a bad transfer, and—if you are unlucky—handed to you by somebody who wants your VM to jump somewhere interesting. The VM's dispatch loop trusts every operand it reads, so something has to check those operands before they get there.

That something is `chunk_verify()` in `src/chunk.c` (and `regchunk_verify()` in `src/regvm.c`), which runs on every chunk and sub-chunk immediately after deserialization and before a single instruction executes.

The verifier makes two passes over each chunk's code:

1. **Structural walk.** Step instruction by instruction, checking that every opcode is real, that its operands do not run past the end of the code, and—recording where each instruction starts as it goes—that the last instruction is a terminator (`OP_HALT`, `OP_RETURN`, or `OP_THROW`). A chunk with no code at all, or a line table shorter than its code, is rejected here.

2. **Operand semantics.** Now that instruction boundaries are known, check that every constant index is in range, that constants used as names really are strings, that sub-chunk references really are sub-chunks, and that every jump—forward, backward, and the skip around a `defer` body—lands exactly on an instruction boundary rather than in the middle of one.

Then it recurses into every closure constant, with nesting capped at 200 so that a maliciously deep chunk tree cannot overflow the C stack during load.

Here is the verifier earning its keep. Each of these is the same 489-byte `greet.latc` with one thing done to it:

```
$ clat tiny.latc          # truncated to 4 bytes
error: file too small for .latc header

$ clat badmagic.latc      # magic overwritten with "XXXX"
error: invalid magic: not a .latc file

$ clat trunc.latc         # first 200 bytes only
error: truncated: incomplete line data

$ clat badop.latc         # one code byte flipped to 0xfe
error: unknown opcode 254 at offset 0

$ clat c13.latc           # one operand byte flipped to 0xf0
error: constant index 240 out of range (const_len=3) at offset 0
```

The register format is checked with the same care, in its own vocabulary: `error: unknown opcode 255 at word 0`.

Verification Is Structural, Not a Checksum

The verifier proves that a file is a well-formed program. It does not prove that it is *your* program. Nothing in either format is hashed or signed, so a byte flip inside constant data sails through untouched:

```
$ clat greet.latc
Hello, Lattice!

$ clat flip.latc          # one byte of the "Hello, " constant changed
Jello, Lattice!
```

If you distribute bytecode over a channel you do not control, verify it the way you would verify any other binary artifact—checksum it, sign it, or both. "The VM will catch it" is true only for corruption that makes the program structurally invalid.

Stack Traces Survive the Round Trip

Chunk names and line tables are serialized, so a failure inside pre-compiled code reads exactly like a failure in source. This program freezes an array and then pushes to it two calls deep:

```
$ clat trace.latc
vm error: cannot call mutating method 'push' on crystal value 'a'
(use thaw(a) to make it mutable)
stack trace (most recent call last):
  [line 11] in <script>
  [line 6] in outer()
  [line 2] in inner()
```

Real function names, real line numbers, identical to running `trace.lat` directly—and identical again for a file produced by the self-hosted compiler.

34.5 The Self-Hosted Compiler

34.5.1 A Compiler Written in Lattice

One of Lattice's most ambitious components lives in `compiler/latc.lat`: a self-hosted compiler that reads `.lat` source code, compiles it to stack VM bytecodes, and writes a `.latc` file—all written in Lattice itself. It is about 5,700 lines long.

```
$ clat compiler/latc.lat greet.lat greet_self.latc
$ clat greet_self.latc
Hello, Lattice!
```

The self-hosted compiler implements the full compilation pipeline: a lexer (using Lattice's built-in `tokenize()` function), a recursive-descent parser, a bytecode emitter, and a binary serializer. It mirrors the structure of the C compiler in `src/stackcompiler.c` but is written entirely in Lattice.

34.5.2 Full Parity

For a long time this chapter would have carried a list of things `latc.lat` could not do. That list is gone. The self-hosted compiler now handles runtime type checks, `test` blocks, nested index assignment at arbitrary depth, variadics, default parameters, and the fast-path opcodes the C compiler emits—and it handles `match`, `select`, and `defer` too:

```
$ clat compiler/latc.lat full.lat full_self.latc
$ clat full_self.latc
measured
12.5664
measured
12
measured
0
got 42
```

That is the same enum-payload match, `defer`, and `select` program from Section 34.1.2, compiled by a compiler written in Lattice, producing output indistinguishable from the C compiler's.

Two gaps remain, and they are gaps in *both* compilers rather than in the self-hosted one: generic syntax parses but does nothing, and there are no map, set, or buffer literals.

34.5.3 Structure of the Self-Hosted Compiler

The file is organized into twelve numbered sections (with a few lettered subsections where a topic outgrew its slot):

1. **Opcode constants**: All stack VM opcodes as integer constants, matching the `Opcode` enum in `include/stackopcode.h` number for number—`OP_CONSTANT` is 0, `OP_ADD` is 8, `OP_CONSTANT_16` is 93, `OP_HALT` is 113. Also the `.latc` constant tags.
2. **Token stream / cursor helpers**: Peeking, advancing, and matching over the token array.

3. **Compiler state**: Global mutable state for the compilation process—the current chunk's code array, constant pool, local variables, scope tracking, and break/continue patching.

4. **Error reporting**.

5. **Emit helpers**: `emit_byte`, `emit_bytes`, `emit_jump`, `emit_loop`, and `emit_wide_op`, which picks between an opcode and its 16-bit sibling.

6. **Constant pool management**: Constants are Lattice arrays shaped `[TAG, ..data]`.

7. **Scope and variable resolution** (plus 7b: `ensure` contract emission, 7c: return type check emission).

8. **Expression parsing**: Precedence climbing (plus 8b: sub-chunk compilation helpers, 8c: `scope` and `select` parsing, 8d: phase system expressions).

9. **Statement parsing**: `let`, `flux`, `fix`, `if`, `for`, `while`, `match`, functions, structs, enums, and more.

10. **Top-level program compilation**.

11. **The .latc serializer**: `serialize_chunk`, `serialize_constant`, `write_u32_le`, `write_i64_le`, and friends.

12. **Main entry point**: Reads the input file, tokenizes, compiles, serializes, and writes the output.

Sections 8b and 8c are the AST-free work of Section 34.1.2 showing up in Lattice source: a sub-chunk compiler and a `select` parser that emits arm records rather than tree pointers.

Listing 34.5: The compiler's main function

```
fn main() {
    let argv = args()
    if argv.len() < 3 {
        eprint("usage: clat compiler/latc.lat <input.lat> <output.latc>")
        exit(1)
    }

    let input_path = argv[1]
    let output_path = argv[2]

    let source = read_file(input_path)
    if source == nil {
        eprint("error: cannot read '" + input_path + "'")
        exit(1)
    }

    tokens = tokenize(source)
    pos = 0

    compile_program()

    let ch_snap = snapshot_chunk()
    let ser_bar = progress.new(1, "Serializing")
    let buf = serialize_latc(ch_snap)
    progress.update(ser_bar, 1)
    progress.finish(ser_bar)

    let ok = write_file_bytes(output_path, buf)
    if !ok {
        eprint("error: cannot write '" + output_path + "'")
        exit(1)
    }
}
```

Notice how the compiler uses Lattice's built-in `tokenize()` function to lex the source code—it does not need to implement its own lexer from scratch. We will explore `tokenize()` in detail in Chapter 36. And notice the progress bars: `latc.lat` is a real command-line tool, so it draws to stderr like one, silently suppressing itself when stderr is not a terminal.

34.5.4 How It Produces .latc Files

The self-hosted compiler builds up the chunk data as Lattice arrays: an array of bytecode values, an array of line numbers, and an array of tagged constants. When compilation is complete, it takes a "snapshot" of the current chunk state and passes it to the serializer.

Listing 34.6: Serialization in the self-hosted compiler

```
fn serialize_chunk(ch: any) {
    // ch = [code_arr, lines_arr, constants_arr, local_names_arr,
    //       chunk_name, fn_has_variadic, default_values]
    let ch_code = ch[0]
    let ch_lines = ch[1]
    let ch_constants = ch[2]
    let ch_local_names = ch[3]

    // Code
    let code_len = ch_code.len()
    write_u32_le(code_len)
    for i in 0..code_len {
        write_u8(ch_code[i])
    }

    // Lines
    let lines_len = ch_lines.len()
    write_u32_le(lines_len)
    for i in 0..lines_len {
        write_u32_le(ch_lines[i])
    }

    // Constants
    let const_len = ch_constants.len()
    write_u32_le(const_len)
    for i in 0..const_len {
        serialize_constant(ch_constants[i])
    }
    // ... local names, chunk name, and call metadata follow
}
```

Why the Serializer Uses a Global Buffer

Look for a buffer parameter in that function and you will not find one. Buffers in Lattice are pass-by-value, so threading one through a recursive serializer would copy the whole accumulated output at every level. Instead `latc.lat` declares `flux ser_buf = nil` at module scope and every `write_*` helper pushes into it.

The same pressure explains the compiler's other unusual shape: chunks are represented as several parallel arrays rather than as a struct, because struct assignment copies too and nested mutation would not propagate. Writing a compiler in Lattice is the most thorough possible interrogation of the language's value semantics.

Not Byte-for-Byte Identical to the C Compiler

The two compilers agree on *behavior*, not on bytes. Compile the same source with each and the files differ in size and in the header:

```
$ clat compile greet.lat                              # C compiler
$ clat compiler/latc.lat greet.lat greet_self.latc    # self-hosted

$ ls -l greet.latc greet_self.latc
-rw-r--r--  489  greet.latc
-rw-r--r--  591  greet_self.latc

$ xxd -l 8 greet_self.latc
00000000: 4c41 5443 0200 0000                      LATC....
```

`latc.lat` writes format version **2**, not 3—it serializes the v2 call metadata but not v3's parameter phase table. Since `LATC_MIN_FORMAT` is 2, the runtime loads its output without complaint, and both files print `Hello, Lattice!`. But if you are diffing artifacts in a build pipeline, compare the two compilers against *themselves*, not against each other.

34.6 Bootstrapping

34.6.1 The Chicken-and-Egg Problem

A self-hosted compiler is a compiler for language L written in language L. This creates a delightful paradox: to run the compiler, you need a Lattice implementation, but to create a Lattice implementation, you need a compiler.

Bootstrapping

Bootstrapping is the process of compiling a self-hosted compiler using an existing implementation of the language. The existing implementation (the "bootstrap compiler") runs the self-hosted compiler's source code, producing a compiled artifact that can then compile itself.

Lattice solves this the same way most self-hosted languages do: with a *bootstrap chain*. The C-based compiler (`src/stackcompiler.c`) serves as the bootstrap implementation. Here is how the chain works:

1. The C-based Lattice implementation (`clat`) can compile and run any `.lat` file.
2. We run the self-hosted compiler (`compiler/latc.lat`) *using* `clat`:

   ```
   clat compiler/latc.lat input.lat output.latc
   ```

3. The self-hosted compiler reads `input.lat`, compiles it to bytecodes, and writes `output.latc`.
4. The resulting `output.latc` can be run by `clat` without any source code.

The interesting part: the self-hosted compiler can also compile *itself*.

```
# Stage 1: the C runtime runs latc.lat over latc.lat
clat compiler/latc.lat compiler/latc.lat stage1.latc

# Stage 1 is now a pre-compiled compiler. Use it on other programs:
clat stage1.latc some_program.lat some_program.latc

# Stage 2: use it to compile the compiler's source again
clat stage1.latc compiler/latc.lat stage2.latc
```

Now compare `stage1.latc` and `stage2.latc`. Stage 1 was produced by the compiler running as *source* under the C runtime; stage 2 was produced by the compiler running as *bytecode*. If the compiler is correct, those two paths have to agree, byte for byte.

34.6.2 The Fixed Point

They do, and it is checked in CI:

```
$ make test-bootstrap
bootstrap:stage1...           PASS
bootstrap:stage2...           PASS
bootstrap:fixed-point...      PASS

Bootstrap fixed point reached: stage1 == stage2
```

Bootstrap Fixed Point

A self-hosted compiler has reached a *fixed point* when compiling its own source with stage n produces a stage $n + 1$ that is byte-identical to stage n. Further iterations cannot change anything, because the input and the compiler are both already stable. It is the strongest cheap evidence you can get that a self-hosted compiler is internally consistent.

Each stage takes a couple of seconds on a modern laptop and produces a 255,511-byte artifact—the compiler, compiled by itself, with nothing left over. If the two stages ever diverged, it would mean the compiler emits different code depending on whether it is itself running as source or as bytecode: a bug in exactly the machinery this chapter describes, caught before anyone had to notice it in the wild.

34.6.3 Why Self-Hosting Matters

Self-hosting is more than a party trick. It provides concrete benefits:

- **Dogfooding**: The compiler exercises a huge slice of the language: arrays, maps, closures, string manipulation, file I/O, control flow, and even the built-in `tokenize()` function. Bugs in any of these features will cause the compiler to fail, providing an excellent integration test.
- **Portability**: The self-hosted compiler is pure Lattice with no C dependencies. It runs anywhere the Lattice runtime runs, including in environments where you cannot compile C code.
- **Extensibility**: Adding new compilation features (optimizations, new opcodes) can be done in Lattice rather than C, with a shorter development loop.
- **Proving the language**: A language that can compile itself has demonstrated a certain level of completeness and expressiveness.

One Backend, One Format Version

The self-hosted compiler targets only the stack VM—there is not a single mention of `RLAT` anywhere in `latc.lat`. Register VM compilation is available only through `clat compile --regvm`. And as we saw in Section 34.5.2, it writes format version 2 rather than 3. Both are natural places for a contributor to start: a register backend, or the v3 phase table.

34.6.4 The Bootstrap in Practice

For day-to-day development, most Lattice users will never need to think about bootstrapping. The C-based `clat` is the standard way to compile and run programs. The self-hosted compiler exists as:

- A comprehensive integration test for the language—and a demanding one, since it leans hard on arrays, closures, string manipulation, file I/O, and value semantics all at once.
- A substantial Lattice program to read—about 5,700 lines.
- A proof that Lattice is expressive enough to write a compiler.
- A starting point for anyone who wants to hack on the compilation pipeline in Lattice rather than C.

Exercises

1. Compile a Lattice program to `.latc` using `clat compile`. Inspect the first 8 bytes of the resulting file with a hex editor or `xxd`. Verify that you see the `LATC` magic and format version 3. Then compile the same program with the self-hosted compiler and check the version byte again—what changed, and why does the file still run?
2. Write a program with at least three functions. Compile it to both `.latc` and `.rlat` and compare the file sizes. The register file is probably the *smaller* of the two. Work out where the stack format spends its extra bytes. (Hint: count line-table entries, not instructions.)
3. Use the self-hosted compiler to compile a program that uses `match`, `defer`, and `select`:

   ```
   clat compiler/latc.lat my_program.lat my_program.latc
   ```

   Run the result with `clat my_program.latc` and verify the output matches running the source directly—including the ordering of the `defer` output.
4. (Challenge) Reach the fixed point by hand. Use `clat` to compile the self-hosted compiler into `stage1.latc`, then use `stage1.latc` to compile the compiler's source again into `stage2.latc`. Are they identical? Now introduce a harmless-looking change to `latc.lat`—reorder two constants, say—and repeat. Does the fixed point still hold?

5. Break a bytecode file on purpose. Take a working `.latc` file and (a) truncate it, (b) flip one byte of the code section, (c) flip one byte inside a string constant. Predict which of the three the verifier will catch before running any of them, then check yourself. What does the one that gets through tell you about what `clat` does and does not promise?
6. The stack format stores a line number for every bytecode *byte*—even for multi-byte instructions where only the first byte really needs one. The register format stores one per instruction. Estimate how much space the stack format spends on line data as a percentage of total file size for a program of your own. Could a run-length encoding scheme close the gap?

What's Next We have seen how Lattice compiles and serializes its bytecodes. But what if you need to extend the language with capabilities written in C—image processing, database access, or hardware integration? In Chapter 35, we explore Lattice's native extension API: how to build shared libraries that plug into the runtime, the value constructors and accessors available to extension authors, and the collection of extensions that ship with Lattice.

Chapter 35

Native Extensions

Lattice's standard library covers a broad range of tasks—files, networking, JSON, cryptography. But eventually you will hit a wall: a C library for image processing, a database driver, or a hardware interface that does not exist in pure Lattice. Native extensions bridge that gap. They let you write performance-critical or platform-specific code in C, compile it into a shared library, and call it seamlessly from Lattice.

In this chapter we will learn the extension API, build an extension from scratch, survey the extensions that ship with Lattice, and understand how the runtime discovers and loads shared libraries.

35.1 The Extension API

35.1.1 The Contract

Every native extension is a shared library (`.dylib` on macOS, `.so` on Linux, `.dll` on Windows) that exports exactly one symbol: `lat_ext_init`. When Lattice loads your extension, it calls this function, passing an opaque `LatExtContext`. Your job is to call `lat_ext_register` to register each function you want to expose:

Extension Initialization

Every Lattice extension must export a function with the signature `void lat_ext_init(LatExtContext *ctx)`. Inside this function, call `lat_ext_register(ctx, name, fn)` once for each function you want to make available to Lattice code.

The entire public API lives in a single header: `include/lattice_ext.h`. Extensions compile against this header only—the internal `LatValue` layout is hidden behind opaque `LatExtValue` pointers, so your extension code never depends on Lattice internals.

35.1.2 Loading an Extension from Lattice

From Lattice code, you load an extension with `require_ext()`:

Listing 35.1: Loading a native extension

```
let sqlite = require_ext("sqlite")

// Bind the functions we need to ordinary names
let db_open  = sqlite["open"]
let db_query = sqlite["query"]
let db_close = sqlite["close"]

let conn = db_open("my_database.db")
let rows = db_query(conn, "SELECT name, age FROM users WHERE age > ?", [21])

for row in rows {
  print("${row["name"]} is ${row["age"]} years old")
}

db_close(conn)
```

The call `require_ext("sqlite")` returns a `Map` where each key is a function name (like `"open"`, `"query"`, `"close"`) and each value is a callable native closure. Once you have pulled a closure out of the map, you call it exactly like any other Lattice function.

Bind First, Then Call

Lattice's grammar does not allow a call directly on an index expression. Writing `sqlite["query"](conn, sql)` is a *parse* error:

```
parse error: expected ',', got '('
```

Bind the closure to a name first (`let db_query = sqlite["query"]`), then call the name. This is the idiom every extension example in this chapter follows, and it reads better anyway—the binding tells you what the function does in your program's vocabulary rather than the library's.

Map-Based Namespacing

Extensions return a `Map` rather than injecting globals. This keeps the global namespace clean and makes it explicit which extension a function came from. One caveat when you pick names: a binding like `let exec = sqlite["exec"]` collides with the built-in `exec()` process function, and the tree-walk interpreter resolves the built-in first. Prefix your bindings (`db_exec`, `ws_send`) and the ambiguity disappears.

35.1.3 The LatExtFn Signature

Every registered extension function has the same C signature:

```
typedef LatExtValue *(*LatExtFn)(LatExtValue **args, size_t argc);
```

The function receives an array of `LatExtValue` pointers (one per argument) and the argument count. It returns a heap-allocated `LatExtValue`, or `NULL` to return `nil`.

Here is a minimal extension function:

```
static LatExtValue *my_add(LatExtValue **args, size_t argc) {
    if (argc < 2) return lat_ext_error("add() expects 2 arguments");
    int64_t a = lat_ext_as_int(args[0]);
    int64_t b = lat_ext_as_int(args[1]);
    return lat_ext_int(a + b);
}
```

35.2 Value Constructors and Accessors

The extension API provides a complete set of constructors and accessors for working with Lattice values from C. All values are wrapped in an opaque `LatExtValue` pointer—you never touch raw `LatValue` structs.

35.2.1 Constructors

All constructors return a heap-allocated `LatExtValue*`. The Lattice runtime deep-clones the value when receiving it, so you do not need to worry about the original being freed.

Error Handling

Returning `lat_ext_error("message")` from an extension function causes Lattice to raise a runtime error. The error message is available in `try`/`catch` blocks just like any other error.

Table 35.1: Extension value constructors

Function	Returns	Description
`lat_ext_int(v)`	`LatExtValue*`	Create an integer value
`lat_ext_float(v)`	`LatExtValue*`	Create a float value
`lat_ext_bool(v)`	`LatExtValue*`	Create a boolean value
`lat_ext_string(s)`	`LatExtValue*`	Create a string (copies the input)
`lat_ext_nil()`	`LatExtValue*`	Create a nil value
`lat_ext_array(elems, len)`	`LatExtValue*`	Create an array from an element array
`lat_ext_map_new()`	`LatExtValue*`	Create an empty map
`lat_ext_error(msg)`	`LatExtValue*`	Create an error value (triggers a Lattice error)

35.2.2 Type Queries

Before accessing a value's contents, check its type:

```
LatExtType lat_ext_type(const LatExtValue *v);
```

The `LatExtType` enum includes: `LAT_EXT_INT`, `LAT_EXT_FLOAT`, `LAT_EXT_BOOL`, `LAT_EXT_STRING`, `LAT_EXT_ARRAY`, `LAT_EXT_MAP`, `LAT_EXT_NIL`, and `LAT_EXT_OTHER` for types not directly representable (like closures or structs).

35.2.3 Accessors

Table 35.2: Extension value accessors

Function	Returns	Description
`lat_ext_as_int(v)`	`int64_t`	Extract integer value
`lat_ext_as_float(v)`	`double`	Extract float value
`lat_ext_as_bool(v)`	`bool`	Extract boolean value
`lat_ext_as_string(v)`	`const char*`	Extract string pointer (borrowed)
`lat_ext_array_len(v)`	`size_t`	Get array length
`lat_ext_array_get(v, i)`	`LatExtValue*`	Get array element (deep clone)
`lat_ext_map_get(v, key)`	`LatExtValue*`	Get map value by key
`lat_ext_map_set(m, k, v)`	`void`	Set a map entry

Memory Management

Values returned by `lat_ext_array_get` and `lat_ext_map_get` are heap-allocated deep clones. You must call `lat_ext_free(v)` on them when you are done. The string returned by `lat_ext_as_string` is borrowed—do *not* free it.

35.2.4 Cleanup

Always free extension values you create for intermediate use:

```
void lat_ext_free(LatExtValue *v);
```

The return value of your extension function is freed automatically by the runtime after it has been cloned into the VM's value space. But any intermediate values you create (e.g., when iterating over an array argument) must be freed by your code.

35.3 Building a .dylib/.so Extension

Let us build a complete extension from scratch. We will create a `"math_extra"` extension that provides a fast integer exponentiation function.

35.3.1 Step 1: Write the C Code

Create a file called `math_extra.c`:

```
#include "lattice_ext.h"

void lat_ext_init(LatExtContext *ctx);

static LatExtValue *math_pow(LatExtValue **args, size_t argc) {
    if (argc < 2)
        return lat_ext_error("pow() expects 2 arguments");
    if (lat_ext_type(args[0]) != LAT_EXT_INT ||
        lat_ext_type(args[1]) != LAT_EXT_INT)
        return lat_ext_error("pow() expects integer arguments");

    int64_t base = lat_ext_as_int(args[0]);
    int64_t exp  = lat_ext_as_int(args[1]);
    int64_t result = 1;

    for (int64_t i = 0; i < exp; i++) {
        result *= base;
```

```
    }
    return lat_ext_int(result);
}

static LatExtValue *math_is_prime(LatExtValue **args, size_t argc) {
    if (argc < 1)
        return lat_ext_error("is_prime() expects 1 argument");
    int64_t n = lat_ext_as_int(args[0]);
    if (n < 2) return lat_ext_bool(false);
    for (int64_t i = 2; i * i <= n; i++) {
        if (n % i == 0) return lat_ext_bool(false);
    }
    return lat_ext_bool(true);
}

void lat_ext_init(LatExtContext *ctx) {
    lat_ext_register(ctx, "pow",      math_pow);
    lat_ext_register(ctx, "is_prime", math_is_prime);
}
```

35.3.2 Step 2: Write a Makefile

The build process is straightforward—compile the C file into a shared library, linking against the Lattice extension header:

```
CC     = cc
CFLAGS = -std=c11 -Wall -Wextra -fPIC -I../../include

UNAME_S := $(shell uname -s)
ifeq ($(UNAME_S),Darwin)
    SHARED_EXT = .dylib
    SHARED_FLAGS = -dynamiclib -undefined dynamic_lookup
else
    SHARED_EXT = .so
    SHARED_FLAGS = -shared
endif

TARGET = math_extra$(SHARED_EXT)

all: $(TARGET)
```

```
$(TARGET): math_extra.c
 $(CC) $(CFLAGS) $(SHARED_FLAGS) -o $@ $<

install: $(TARGET)
 @mkdir -p $(HOME)/.lattice/ext
 cp $(TARGET) $(HOME)/.lattice/ext/

clean:
 rm -f $(TARGET)
```

Key details:

- `-fPIC` generates position-independent code required for shared libraries.
- `-I../../include` points to the Lattice include directory where `lattice_ext.h` lives.
- On macOS, `-dynamiclib -undefined dynamic_lookup` creates a `.dylib` that resolves Lattice symbols at load time (they are provided by the `clat` binary).
- On Linux, `-shared` creates a `.so`.

35.3.3 Step 3: Build and Install

Listing 35.2: Building and installing an extension

```
$ make
$ make install    # copies math_extra.dylib to ~/.lattice/ext/
```

Installing into `~/.lattice/ext/` matters more than it used to. As we will see in Section 35.5, that directory is the *first* place the runtime looks, and a library sitting in your project's `./extensions/` will not be found at all unless you opt in explicitly.

35.3.4 Step 4: Use It from Lattice

Listing 35.3: Using the math_extra extension

```
let math = require_ext("math_extra")

let int_pow  = math["pow"]
let is_prime = math["is_prime"]

print(int_pow(2, 10))       // 1024
print(int_pow(3, 5))        // 243
print(is_prime(17))         // true
print(is_prime(100))        // false
```

Assuming you installed the library into `~/.lattice/ext/`, that is the whole ceremony:

Listing 35.4: Running the example

```
$ clat use_math.lat
1024
243
true
false
```

Returning Compound Values

Extensions can return arrays and maps, not just scalars. Use `lat_ext_map_new()` and `lat_ext_map_set()` to build structured results—database rows, image metadata, or configuration objects.

35.4 Available Extensions

Lattice ships with six extensions in the `extensions/` directory. Each is a self-contained C project with its own `Makefile`.

35.4.1 ffi — Foreign Function Interface

The FFI extension lets you call functions from arbitrary shared libraries at runtime—no C code required. It supports struct marshalling, callback trampolines, memory operations, and type signatures.

Listing 35.5: Using the FFI extension

```
let ffi = require_ext("ffi")

let ffi_open  = ffi["open"]
let ffi_sym   = ffi["sym"]
let ffi_call  = ffi["call"]
let ffi_close = ffi["close"]

// Open the C math library
let libm = ffi_open("/usr/lib/libm.dylib")

// "f>f": one double argument, returns a double
let sqrt_fn = ffi_sym(libm, "sqrt", "f>f")

// Call it
let result = ffi_call(sqrt_fn, 144.0)
print(result) // 12

ffi_close(libm)
```

Reading FFI Signatures

A signature is a string of single-character type codes: argument types, a `>`, then the return type. `"ii>i"` is "two 64-bit ints in, one 64-bit int out"; `"f>f"` is "one `double` in, one `double` out"; omitting the `>` means the function returns `void`.
The codes are worth memorising because they are not the ones you would guess: `i` is `int64_t`, `b`/`w`/`d` are 8/16/32-bit signed integers, `f` is `double` while `g` is a single-precision `float`, `s` is `const char *`, `p` is `void *`, `z` is `size_t`, `c` is a plain C `int`, and `v` is `void` (return position only). Uppercase `B`/`W`/`D` and `u` are the unsigned widths. The full list lives in the header comment of `extensions/ffi/ffi.c`.

Note that `print` shows the result as `12`, not `12.0`—Lattice prints floats without a trailing zero fraction. The value really is a `Float`; `typeof(result)` will tell you so.

Core functions include `open`, `close`, `sym`, `call`, `addr`, `errno`, `error`, `nullptr`, and `strerror`. Struct marshalling is available via `struct_define`, `struct_alloc`, `struct_get`, `struct_set`, `struct_free`, `struct_to_map`, and `struct_from_map`. Memory operations include `alloc`, `free`, `memcpy`, `memset`, `read_i32`, `write_i32`, and friends, plus `callback` and `callback_free` for handing a Lattice closure to C as a function pointer.

35.4.2 sqlite — SQLite Database

A complete SQLite client with parameterized queries, transactions, and metadata:

Listing 35.6: Using the SQLite extension

```
let sqlite = require_ext("sqlite")

let db_open  = sqlite["open"]
let db_exec  = sqlite["exec"]
let db_query = sqlite["query"]
let db_close = sqlite["close"]

let conn = db_open("app.db")
db_exec(conn, "CREATE TABLE IF NOT EXISTS users (name TEXT, age INT)")
db_exec(conn, "INSERT INTO users VALUES (?, ?)", ["Alice", 30])
db_exec(conn, "INSERT INTO users VALUES (?, ?)", ["Bob", 25])

let rows = db_query(conn, "SELECT * FROM users WHERE age > ?", [20])
for row in rows {
  print("${row["name"]}: ${row["age"]}")
}
// Alice: 30
// Bob: 25

db_close(conn)
```

Functions: `open`, `close`, `query`, `exec`, `status`, `last_insert_rowid`.

35.4.3 pg — PostgreSQL

A PostgreSQL client built on `libpq`:

Listing 35.7: Using the PostgreSQL extension

```
let pg = require_ext("pg")

let pg_connect = pg["connect"]
let pg_query   = pg["query"]
let pg_close   = pg["close"]

let conn = pg_connect("host=localhost dbname=myapp")
let rows = pg_query(conn, "SELECT id, email FROM accounts LIMIT 10")

for row in rows {
  print("${row["id"]}: ${row["email"]}")
}

pg_close(conn)
```

Functions: `connect`, `close`, `query`, `exec`, `status`.

35.4.4 redis — Redis Client

A minimal Redis client that implements the RESP protocol over raw TCP sockets—no external dependencies:

Listing 35.8: Using the Redis extension

```
let redis = require_ext("redis")

let r_connect = redis["connect"]
let r_get     = redis["get"]
let r_set     = redis["set"]
let r_incr    = redis["incr"]
let r_close   = redis["close"]

let conn = r_connect("127.0.0.1", 6379)
r_set(conn, "greeting", "Hello from Lattice!")
let val = r_get(conn, "greeting")
print(val) // Hello from Lattice!

r_incr(conn, "counter")
r_incr(conn, "counter")
print(r_get(conn, "counter")) // 2

r_close(conn)
```

Functions: `connect`, `close`, `command`, `get`, `set`, `del`, `exists`, `expire`, `keys`, `incr`, `lpush`, `lrange`, `publish`, `ping`.

35.4.5 image — Image Metadata and Operations

Reads image headers directly (PNG, JPEG, GIF, BMP, WebP) without external library dependencies. Uses macOS `sips` for resize, convert, and thumbnail operations:

Listing 35.9: Using the image extension

```
let image = require_ext("image")

let img_info   = image["info"]
let img_resize = image["resize"]

let info = img_info("photo.png")
print("Format: ${info["format"]}")
print("Size: ${info["width"]}x${info["height"]}")
// Format: png
// Size: 1481x728

// Resize (macOS only, uses sips):
img_resize("photo.png", 800, 600, "photo_small.png")
```

Functions: `info`, `format`, `dimensions`, `resize`, `convert`, `thumbnail`.

35.4.6 websocket — WebSocket Client and Server

Implements RFC 6455 WebSocket framing with support for both client and server roles, ping/pong, fragmented messages, and timeouts:

Listing 35.10: Using the WebSocket extension

```
let ws = require_ext("websocket")

let ws_connect = ws["connect"]
let ws_send    = ws["send"]
let ws_recv    = ws["recv"]
let ws_close   = ws["close"]

// Client connection
let conn = ws_connect("ws://echo.websocket.org")
ws_send(conn, "Hello, WebSocket!")
let response = ws_recv(conn)
print(response) // Hello, WebSocket!
ws_close(conn)
```

Functions include `connect`, `connect_auto` (auto-reconnect), `listen`, `accept`, `server_close`, `send`, `recv`, `recv_timeout`, `send_binary`, `close`, `status`, `ping`, and `set_timeout`.

35.5 Extension Search Paths

When you call `require_ext("name")`, the runtime searches for the shared library in the following order:

1. `~/.lattice/ext/name.dylib` (or `.so` on Linux, `.dll` on Windows) — the user's global extension directory.

2. `$LATTICE_EXT_PATH/name.dylib` — a directory named by the `LATTICE_EXT_PATH` environment variable.

3. `./extensions/name.dylib` and `./extensions/name/name.dylib` — the project-local extensions directory. **Only searched when `LATTICE_EXT_ALLOW_CWD` is set in the environment**, and even then it is tried last.

The first match wins. If no library is found, `require_ext` raises a runtime error naming the places it looked:

Listing 35.11: require_ext failure message

```
require_ext: cannot find extension 'math_extra'
  (searched ~/.lattice/ext/, $LATTICE_EXT_PATH)
```

With `LATTICE_EXT_ALLOW_CWD` set, the message grows a third entry:

```
, ./extensions/ (LATTICE_EXT_ALLOW_CWD)
```

The Search Order Changed in v0.6.4

Earlier releases searched the working directory's `./extensions/` *first*. The hardening work landed during the v0.6.2 development window and first shipped in **v0.6.4**, which is the release where the behavior changes for anyone running a tagged build: from v0.6.4 onward, `./extensions/` is gated behind `LATTICE_EXT_ALLOW_CWD` and demoted to last place. If you are on v0.5.0, v0.6.0, or v0.6.1, the old implicit search is still in force and nothing has broken yet—which is exactly why it is worth setting the variable explicitly before you upgrade. Programs that relied on picking an extension up out of `./extensions/` now fail with the "cannot find extension" error until you either install the library or opt in:

```
LATTICE_EXT_ALLOW_CWD=1 clat my_program.lat
```

The reason is library planting. If the working directory is searched implicitly, anyone who can drop a file into the directory you happen to `cd` into can get their C code loaded and executed by your interpreter. Trusted, user-owned directories now come first, and the untrusted one is opt-in. You can see the whole search construction in `src/ext.c`.

Listing 35.12: Extension search path behavior

```
# Installed globally -- found at path #1
~/.lattice/ext/sqlite.dylib

# Pointed at a shared team directory -- found at path #2
LATTICE_EXT_PATH=/opt/lattice/ext clat my_program.lat

# Built in-tree, opted in explicitly -- found at path #3
LATTICE_EXT_ALLOW_CWD=1 clat my_program.lat
```

Global Installation

Each extension's `Makefile` includes a `make install` target that copies the built library to `~/.lattice/ext/`—which is now the first place the runtime looks. This makes the extension available to all your Lattice projects with no environment variables at all:

```
cd extensions/sqlite && make && make install
```

35.5.1 Security Considerations

Native extensions execute arbitrary C code with full process permissions. The runtime validates that the extension name does not contain path separators (`/` or `\`) to prevent path traversal, and it prefers trusted directories over the working directory, but it does not sandbox the loaded code.

Trust Your Extensions

Only load extensions from sources you trust. A malicious extension has the same access as the Lattice process itself—it can read files, make network connections, or modify memory.

35.5.2 WASM Limitation

When Lattice is compiled to WebAssembly (via Emscripten), native extensions are not available. The `require_ext` function raises a catchable error explaining that dynamic loading is unsupported in the WASM environment:

Listing 35.13: require_ext under WebAssembly

```
require_ext: native extensions not available in WASM (tried 'sqlite')
```

Exercises

1. Build the `math_extra` extension described in Section 35.3. Load it from a Lattice program and verify that `pow(2, 20)` returns `1048576`.
2. Add a `gcd` function to the `math_extra` extension that computes the greatest common divisor of two integers using Euclid's algorithm. Register it in `lat_ext_init` and test it from Lattice.
3. Write an extension function that accepts a Lattice array of strings and returns them sorted alphabetically using C's `qsort`. Use `lat_ext_array_len`, `lat_ext_array_get`, and `lat_ext_as_string` to extract the strings, sort them, and return a new array via `lat_ext_array`.
4. The extension API uses opaque `LatExtValue` pointers to hide the internal `LatValue` layout. What are the advantages of this design over exposing the raw struct? What are the disadvantages (hint: think about performance)?
5. (Challenge) Write an extension that wraps a third-party C library of your choice (e.g., `zlib` for compression, `curl` for HTTP requests). Include proper error handling for all failure cases.

What's Next We have seen how to extend Lattice with C code. But Lattice also provides powerful tools for inspecting and manipulating programs from *within* the language itself. In Chapter 36, we explore runtime code introspection, the `tokenize()` function, struct reflection, and the dynamic features that make Lattice a surprisingly flexible metaprogramming platform.

Chapter 36

Metaprogramming and Reflection

Most of the time, a program manipulates data: numbers, strings, arrays, structs. But sometimes a program needs to manipulate *itself*—inspect its own structure, examine types at runtime, build code from strings, or interrogate a struct to discover what fields it carries. These capabilities fall under the umbrella of **metaprogramming** and **reflection**.

Lattice is not a "macro language" in the tradition of Lisp or Rust. It does not have compile-time code generation or hygienic macros. Instead, it offers a pragmatic set of *runtime* introspection tools: functions that let you peek inside values, tokenize source code, convert between structs and maps, and even evaluate strings as Lattice code on the fly.

These tools are powerful, and like any powerful tool, they come with trade-offs. Let's explore them.

36.1 Runtime Code Introspection and Dynamic Features

Lattice provides a family of built-in functions for examining values at runtime. These are not methods on objects—they are global functions available everywhere, defined in `src/eval.c` (for the tree-walk interpreter) and mirrored in the bytecode VM dispatch loops.

36.1.1 Type Queries

The most fundamental introspection is asking "what *is* this value?"

Listing 36.1: Querying types at runtime

```
print(typeof(42))           // Int
print(typeof(3.14))         // Float
print(typeof("hello"))      // String
print(typeof(true))         // Bool
print(typeof([1, 2, 3]))    // Array
print(typeof(nil))          // Nil
```

The `typeof` function returns a string naming the value's type. Under the hood, it calls `builtin_typeof_str()` from `src/builtins.c`, which is a straightforward `switch` on the value's internal type tag. The possible return values are: `"Int"`, `"Float"`, `"Bool"`, `"String"`, `"Array"`, `"Struct"`, `"Closure"`, `"Unit"`, `"Nil"`, `"Range"`, `"Map"`, `"Channel"`, `"Buffer"`, `"Tuple"`, `"Set"`, `"Iterator"`, `"Ref"`, and `"Enum"`.

typeof()

`typeof(val)` returns a `String` naming the runtime type of `val`. For structs, it returns `"Struct"` (not the struct's name—use `struct_name()` for that).

This makes `typeof` useful for writing generic functions that need to branch on type:

Listing 36.2: A generic serializer using typeof

```
fn serialize(val: any) -> String {
  let t = typeof(val)
  if t == "Int" || t == "Float" || t == "Bool" {
    return to_string(val)
  }
  if t == "String" {
    return "\"" + val + "\""
  }
  if t == "Array" {
    let parts = val.map(|item| serialize(item))
    return "[" + parts.join(", ") + "]"
  }
  if t == "Nil" {
    return "null"
  }
  return repr(val)
}

print(serialize([1, "hello", true]))
// Output: [1, "hello", true]
```

36.1.2 Phase Queries

Lattice's phase system distinguishes fluid (mutable) values from crystallized (frozen) ones. The `phase_of` function lets you check a value's phase at runtime:

Listing 36.3: Querying value phase

```
flux temperature = 72.0
print(phase_of(temperature))     // fluid

fix pi = 3.14159
print(phase_of(pi))              // crystal

let config = freeze(["debug", "verbose"])
print(phase_of(config))          // crystal
```

The return values are `"fluid"`, `"crystal"`, or `"unphased"` (for values that have not been explicitly assigned a phase). This is particularly useful when writing library code that must respect its callers' mutability constraints:

Listing 36.4: Respecting caller's phase constraints

```
fn safe_append(collection: any, item: any) -> any {
  if phase_of(collection) == "crystal" {
    // Don't mutate a frozen collection; return a new one
    let thawed = thaw(collection)
    thawed.push(item)
    return freeze(thawed)
  }
  collection.push(item)
  return collection
}
```

36.1.3 String Representations

Two functions convert values to strings, each with a different purpose:

- `to_string(val)` — produces a human-readable display string. Strings are returned without quotes, arrays are formatted with brackets, and special values like `nil` become `"nil"`.
- `repr(val)` — produces a representation string that could (in principle) be parsed back into the original value. Strings are quoted, struct fields are shown, and closures display their parameter lists.

Listing 36.5: to_string vs. repr

```
let greeting = "hello"
print(to_string(greeting))   // hello
print(repr(greeting))        // "hello"

let items = [1, 2, 3]
print(to_string(items))      // [1, 2, 3]
print(repr(items))           // [1, 2, 3]

print(to_string(nil))        // nil
print(repr(nil))             // nil
```

When to Use repr()

Use `repr()` when writing debug output or logging. It shows you the "programmer's view" of a value, including type information that `to_string()` strips away. For user-facing output, prefer `to_string()` or string interpolation.

36.1.4 Dynamic Evaluation with lat_eval()

The most powerful (and most dangerous) introspection tool is `lat_eval()`, which takes a string of Lattice source code, parses and evaluates it at runtime, and returns the result:

Listing 36.6: Dynamic code evaluation

```
let result = lat_eval("2 + 3 * 4")
print(result)  // 14

// Define a function dynamically
lat_eval("fn greet(name: String) { print(\"Hello, \" + name) }")
greet("World")  // Hello, World
```

Under the hood, `lat_eval()` in `src/eval.c` does exactly what the top-level interpreter does: it tokenizes the string, parses it into an AST, registers any function and struct declarations into the current evaluator's registries, and executes the statements. Bindings created inside `lat_eval()` persist in the caller's scope.

lat_eval() and Security

`lat_eval()` executes arbitrary code with full access to the runtime environment. Never pass untrusted input (user data, network data) to `lat_eval()`. An attacker could read files, make network connections, or execute system commands. In production code, prefer structured approaches (closures, maps, pattern matching) over `lat_eval()`.

36.1.5 Completeness Checking with is_complete()

A companion to `lat_eval()` is `is_complete()`, which checks whether a source string has balanced brackets and parentheses:

Listing 36.7: Checking source completeness

```
print(is_complete("1 + 2"))        // true
print(is_complete("fn foo() {"))   // false: unclosed brace
print(is_complete("if true { }"))  // true
```

This is used internally by the REPL to determine whether to wait for more input or evaluate the current line. It tokenizes the string, counts bracket depth, and returns `true` if the depth is zero or less.

36.2 tokenize() — Inspecting Tokens

The `tokenize()` built-in function exposes Lattice's own lexer to user code. Given a string of source code, it returns an array of `Token` structs:

Listing 36.8: Tokenizing Lattice source code

```
let tokens = tokenize("let x = 42 + y")

for tok in tokens {
  print("${tok.type}: ${tok.text}")
}
// Output:
// let: let
// identifier: x
// =: =
// integer: 42
// +: +
// identifier: y
```

That output surprises most people the first time. The token *type* is not a shouty enum name like `"IDENT"`—it is the human-readable name the compiler itself uses in its error messages. Keywords and operators are named after themselves: the type of a `let` keyword is the string `"let"`, and the type of a `+` is the string `"+"`. Only the five categories that carry a payload get a category name: `"identifier"`, `"integer"`, `"float"`, `"string"`, and `"#mode"`.

Each element in the returned array is a struct with three fields:

- `type` — a `String` naming the token type (e.g., `"identifier"`, `"integer"`, `"string"`, `"+"`, `"{"`, `"fn"`).

- `text` — a `String` containing the token's text. For identifiers and literals, this is the actual text from the source (string literals arrive *without* their quotes). For operators and keywords, the text repeats the type.

- `line` — an `Int`: the 1-based source line the token came from.

Because the type name is the source spelling, matching on tokens reads almost like reading the code:

Listing 36.9: Token fields in practice

```
let tokens = tokenize("let msg = \"hi\"\nlet pi = 3.5")
for tok in tokens {
  print("${tok.type} | ${tok.text} | ${tok.line}")
}
// let | let | 1
// identifier | msg | 1
// = | = | 1
// string | hi | 1
// let | let | 2
// identifier | pi | 2
// = | = | 2
// float | 3.5 | 2
```

The trailing `EOF` token is stripped from the result, so you only see meaningful tokens. The naming table lives in `src/token.c`, in the `simple_names` array and the `token_type_name()` function just below it—worth a look when you need a type name you have not seen before.

36.2.1 Use Cases for tokenize()

The most prominent use of `tokenize()` is in the self-hosted compiler (see Section 34.5). The compiler written in Lattice (`compiler/latc.lat`) calls `tokenize()` on the input source file to get its token stream, then implements a recursive-descent parser over those tokens. This means the self-hosted compiler does not need to reimplement lexing—it reuses the C lexer via this single function call.

Beyond the self-hosted compiler, `tokenize()` enables:

Listing 36.10: Counting keywords in a source file

```
fn count_keywords(source: String) -> Int {
  let tokens = tokenize(source)
  let keywords = ["fn", "let", "flux", "fix", "if", "else",
                  "for", "while", "return", "match", "struct"]
  flux count = 0
  for tok in tokens {
    for kw in keywords {
      if tok.type == kw {
        count = count + 1
      }
    }
  }
  return count
}

print(count_keywords(
  "fn f(a: Int) -> Int { if a > 0 { return a } else { return 0 } }"))
// 5
```

Notice that we compare against `tok.type`, not `tok.text`. A keyword never reaches us as an identifier—the lexer has already classified it—so a scan that looks for `"identifier"` tokens whose text happens to spell `"fn"` will find nothing at all.

Listing 36.11: A syntax highlighter skeleton

```
fn highlight(source: String) -> String {
  let tokens = tokenize(source)
  flux result = ""
  for tok in tokens {
    if tok.type == "fn" || tok.type == "let" || tok.type == "if" {
      result = result + "\x1b[35m" + tok.text + "\x1b[0m "
    } else if tok.type == "string" {
      result = result + "\x1b[32m\"" + tok.text + "\"\x1b[0m "
    } else if tok.type == "integer" || tok.type == "float" {
      result = result + "\x1b[34m" + tok.text + "\x1b[0m "
    } else {
      result = result + tok.text + " "
    }
  }
  return result
}

print(highlight("let n = 42"))
// (magenta) let  n = (blue) 42
```

tokenize() and the Bytecode VMs

The `tokenize()` function is available in all three backends. In the tree-walk interpreter, it calls `lexer_tokenize()` directly. In the bytecode VMs, it is dispatched as a native built-in function that invokes the same C lexer. The returned tokens are identical regardless of backend.

36.2.2 Token Types

Lattice's lexer produces a rich set of token types. Here is a representative sample:

Every keyword and every operator in the language follows the same rule—type and text are both the source spelling—so the table generalizes without surprises. For a complete list, see Chapter A.

36.3 Struct Reflection Functions

Lattice structs are not opaque—at runtime, you can interrogate them to discover their name, fields, and values. This makes it possible to write generic code that operates on *any* struct, regardless of its definition.

Table 36.1: Common token types returned by tokenize()

Token Type	Example Text	Description
`identifier`	`my_var`	Identifier
`integer`	`42`	Integer literal
`float`	`3.14`	Float literal
`string`	`hello`	String literal (without quotes)
`fn`	`fn`	`fn` keyword
`let`	`let`	`let` keyword
`flux`	`flux`	`flux` keyword
`fix`	`fix`	`fix` keyword
`if`	`if`	`if` keyword
`+`	`+`	+ operator
`=`	`=`	= assignment
`==`	`==`	equality comparison
`->`	`->`	return-type arrow
`{`	`{`	opening brace
`}`	`}`	closing brace

36.3.1 struct_name()

Listing 36.12: Getting a struct's type name

```
struct User {
  name: String,
  age: Int,
}

let alice = User { name: "Alice", age: 30 }
print(struct_name(alice))  // User
```

The `struct_name()` function returns the struct's type name as a string. This is distinct from `typeof(alice)`, which returns `"Struct"` for all struct instances. With `struct_name()` you can distinguish between different struct types at runtime.

36.3.2 struct_fields()

Listing 36.13: Listing a struct's field names

```
struct Point {
  x: Float,
  y: Float,
}

let origin = Point { x: 0.0, y: 0.0 }
let fields = struct_fields(origin)
print(fields)  // [x, y]
```

The `struct_fields()` function returns an array of field name strings. The names are in declaration order—the same order you specified when defining the struct.

36.3.3 struct_to_map()

Perhaps the most useful reflection function, `struct_to_map()` converts a struct instance into a `Map` where the keys are field names and the values are deep clones of the field values:

Listing 36.14: Converting a struct to a map

```
struct Config {
  host: String,
  port: Int,
  debug: Bool,
}

let cfg = Config { host: "localhost", port: 8080, debug: true }
let map = struct_to_map(cfg)

print(map["host"])    // localhost
print(map["port"])    // 8080
print(map["debug"])   // true
```

This is invaluable for serialization. Instead of writing a custom serializer for each struct type, you can convert to a map and pass it to `json_stringify()`:

Listing 36.15: Generic struct serialization via struct_to_map

```
fn to_json(val: any) -> String {
  if typeof(val) == "Struct" {
    let map = struct_to_map(val)
    return json_stringify(map)
  }
  return json_stringify(val)
}

struct Product {
  name: String,
  price: Float,
  in_stock: Bool,
}

let item = Product { name: "Widget", price: 9.99, in_stock: true }
print(to_json(item))
// Output: {"name":"Widget","price":9.99,"in_stock":true}
```

36.3.4 struct_from_map()

The inverse of `struct_to_map()` is `struct_from_map()`, which creates a struct instance from a type name and a map of field values:

Listing 36.16: Creating a struct from a map

```
struct User {
  name: String,
  age: Int,
}

let data = Map::new()
data["name"] = "Bob"
data["age"] = 25

let user = struct_from_map("User", data)
print(user.name)  // Bob
print(user.age)   // 25
```

The struct type must already be defined in the current scope. If a field is missing from the map, it defaults to `nil`. This function is particularly useful for deserialization—parsing JSON into a map, then converting the map into a typed struct:

Listing 36.17: JSON deserialization into a struct

```
struct Task {
  title: String,
  done: Bool,
  priority: Int,
}

fn parse_task(json_str: String) -> any {
  let map = json_parse(json_str)
  return struct_from_map("Task", map)
}

let task = parse_task("{\"title\": \"Write docs\", \"done\": false, \"priority\": 1}")
print(task.title)     // Write docs
print(task.priority)  // 1
```

36.3.5 Combining Reflection for Generic Programming

With these four functions, you can write powerful generic code:

Listing 36.18: A generic equality checker for structs

```
fn struct_equal(a: any, b: any) -> Bool {
  if typeof(a) != "Struct" || typeof(b) != "Struct" {
    return false
  }
  if struct_name(a) != struct_name(b) {
    return false
  }
  let fields_a = struct_fields(a)
  let fields_b = struct_fields(b)
  if len(fields_a) != len(fields_b) {
    return false
  }
  let map_a = struct_to_map(a)
  let map_b = struct_to_map(b)
  for field in fields_a {
    if map_a[field] != map_b[field] {
      return false
    }
  }
  return true
}

struct Point {
  x: Float,
  y: Float,
}

let p1 = Point { x: 1.0, y: 2.0 }
let p2 = Point { x: 1.0, y: 2.0 }
let p3 = Point { x: 3.0, y: 4.0 }

print(struct_equal(p1, p2))  // true
print(struct_equal(p1, p3))  // false
```

Listing 36.19: A generic struct printer

```
fn describe_struct(val: any) -> String {
  if typeof(val) != "Struct" {
    return repr(val)
  }
  let name = struct_name(val)
  let fields = struct_fields(val)
  let map = struct_to_map(val)
  flux parts = []
  for field in fields {
    parts.push(field + ": " + repr(map[field]))
  }
  return name + " { " + parts.join(", ") + " }"
}

struct Color {
  r: Int,
  g: Int,
  b: Int,
}

let coral = Color { r: 255, g: 127, b: 80 }
print(describe_struct(coral))
// Output: Color { r: 255, g: 127, b: 80 }
```

36.4 format() — String Building

While string interpolation with `${}` handles most formatting needs, the `format()` function provides a positional placeholder approach similar to Python's `str.format()` or Rust's `format!()` macro.

36.4.1 Basic Usage

Listing 36.20: Using format() for string building

```
let msg = format("{} is {} years old", "Alice", 30)
print(msg)  // Alice is 30 years old

let report = format("Total: {} items, {} in stock", 150, 42)
print(report)  // Total: 150 items, 42 in stock
```

The format string uses `{}` as a placeholder. Arguments are consumed left to right, one per placeholder. Each argument is converted to its display string (equivalent to calling `to_string()` on it).

36.4.2 Escaping Braces

If you need a literal `{` or `}` in the output, double it:

Listing 36.21: Escaping braces in format strings

```
let json_template = format("{{\"name\": \"{}\", \"age\": {}}}", "Bob", 25)
print(json_template)  // {"name": "Bob", "age": 25}
```

The implementation in `src/format_ops.c` handles this by checking for consecutive `{{` and `}}` patterns, emitting a single brace for each pair.

36.4.3 Error Handling

If you provide fewer arguments than placeholders, `format()` raises a runtime error:

Listing 36.22: format() error on missing arguments

```
// This will raise an error:
// let msg = format("{} and {}", "Alice")
// Error: format: not enough arguments for placeholders
```

Extra arguments are silently ignored—only the placeholders that exist in the format string consume arguments.

36.4.4 format() vs. String Interpolation

When should you use `format()` instead of string interpolation?

- **Dynamic format strings**: When the template is not known at compile time—for instance, loaded from a configuration file or received over a network.
- **Reusable templates**: When you want to define a template once and apply it with different arguments.
- **Array-based arguments**: When the arguments come from a computed source rather than inline variables.

Listing 36.23: Dynamic templates with format()

```
// A logging system with configurable format
flux log_template = "[{}] {}: {}"

fn log(level: String, message: String) {
  let timestamp = to_string(time_ms())
  print(format(log_template, timestamp, level, message))
}

log("INFO", "Server started")
// Output: [1709234567890] INFO: Server started

log("ERROR", "Connection refused")
// Output: [1709234567891] ERROR: Connection refused
```

Prefer Interpolation for Readability

For most cases, string interpolation (`"Hello, ${name}!"`) is more readable than `format("Hello, {}!", name)`. Reserve `format()` for situations where the template itself is dynamic or where you are building strings programmatically.

36.5 The Power and Danger of Runtime Features

Runtime introspection and dynamic evaluation are a double-edged sword. Let's be honest about both edges.

36.5.1 The Power

Dynamic features let you build things that would otherwise require code generation or preprocessors:

Listing 36.24: A configuration-driven plugin system

```
struct Plugin {
  name: String,
  version: String,
  init: fn() -> Bool,
}

fn load_plugin_from_config(config: Map) -> any {
  let name = config["name"]
  let source = read_file("plugins/" + name + ".lat")
  lat_eval(source)
  // The plugin registers itself via a global function
  return true
}
```

Listing 36.25: A test runner that discovers tests

```
fn find_test_functions(source: String) -> Array {
  let tokens = tokenize(source)
  flux test_names = []
  for i in 0..len(tokens) - 1 {
    if tokens[i].type == "TEST" {
      if i + 1 < len(tokens) && tokens[i + 1].type == "STRING_LIT" {
        test_names.push(tokens[i + 1].text)
      }
    }
  }
  return test_names
}

let source = read_file("my_tests.lat")
let tests = find_test_functions(source)
print("Found " + to_string(len(tests)) + " tests:")
for name in tests {
  print("  - " + name)
}
```

Listing 36.26: A generic ORM-style mapper

```
fn map_rows_to_structs(rows: Array, struct_type: String) -> Array {
  flux results = []
  for row in rows {
    let instance = struct_from_map(struct_type, row)
    results.push(instance)
  }
  return results
}

struct Employee {
  name: String,
  department: String,
  salary: Float,
}

// Imagine these rows came from a database query
let rows = [
  Map::new(),
  Map::new(),
]
rows[0]["name"] = "Alice"
rows[0]["department"] = "Engineering"
rows[0]["salary"] = 95000.0
rows[1]["name"] = "Bob"
rows[1]["department"] = "Marketing"
rows[1]["salary"] = 85000.0

let employees = map_rows_to_structs(rows, "Employee")
for emp in employees {
  print("${emp.name} in ${emp.department}")
}
// Output:
// Alice in Engineering
// Bob in Marketing
```

36.5.2 The Dangers

Every runtime feature trades away some combination of safety, performance, and readability.

Performance. `lat_eval()` must lex, parse, and interpret the string on every call. There is no caching and no compilation—it is a full interpreter invocation. In a tight loop, this overhead is devastating.

Security. `lat_eval()` on untrusted input is a code injection vulnerability. The evaluated code runs with the same privileges as the host program.

Debuggability. Dynamically generated code does not have source file locations. Error messages will reference the evaluated string, not a file and line number. Stack traces from `lat_eval()` are harder to read.

Type safety. Reflection functions like `struct_to_map()` erase type information. The resulting map has string keys and `any`-typed values—you lose the compiler's ability to check field names and types.

Refactoring. Code that uses `lat_eval()` to define functions or structs dynamically is invisible to static analysis tools, formatters, and documentation generators. Renaming a function in your editor will not catch references inside evaluated strings.

The lat_eval() Rule of Thumb

If you find yourself reaching for `lat_eval()`, stop and ask: "Can I solve this with closures, maps, or pattern matching instead?" The answer is almost always yes. Reserve `lat_eval()` for genuinely dynamic scenarios: plugin systems, REPLs, code generators, and development tools.

36.5.3 Guidelines for Responsible Metaprogramming

1. **Prefer closures over eval.** Instead of building a function as a string and evaluating it, accept a closure parameter and call it.

2. **Use struct reflection for serialization, not for business logic.** `struct_to_map()` and `struct_from_map()` are ideal for converting between data formats. They are less ideal for implementing core application behavior.

3. **Validate before evaluating.** If you must use `lat_eval()`, validate the input first. At minimum, check that it tokenizes without errors using `is_complete()`.

4. **Use tokenize() for analysis, not execution.** `tokenize()` is safe—it only reads, never evaluates. It is the right tool for syntax highlighting, code analysis, and documentation generation.

5. **Document dynamic behavior.** When your code uses reflection or dynamic evaluation, add comments explaining *why* the dynamic approach is necessary and what the expected inputs are.

Listing 36.27: Closures instead of eval: the better way

```
// BAD: building code as strings
// let code = "fn process(x: Int) -> Int { return x * " + to_string(multiplier) + " }"
// lat_eval(code)

// GOOD: use a closure that captures the value
fn make_processor(multiplier: Int) -> fn(Int) -> Int {
  return |x| x * multiplier
}

let process = make_processor(3)
print(process(10))  // 30
```

36.6 Exercises

1. Write a function `pretty_print(val: any)` that produces a formatted, indented string representation of any value. For structs, use `struct_name()` and `struct_to_map()` to show the struct type and fields. For arrays and maps, indent nested elements.
2. Use `tokenize()` to write a function that counts the number of functions defined in a Lattice source file. Look for the `FN` token followed by an `IDENT` token.
3. Write a `diff_structs(a: any, b: any)` function that compares two struct instances of the same type and returns an array of field names that differ. Use `struct_fields()` and `struct_to_map()` to implement it generically.
4. Create a simple calculator REPL that reads a line from the user with `input()`, evaluates it with `lat_eval()`, and prints the result. Add basic error handling: catch evaluation errors and display a helpful message instead of crashing.
5. (Challenge) Write a "struct validator" function that takes a struct instance and a map of `{field_name: validation_closure}` pairs. For each field, call the corresponding validation closure with the field's value and collect any failures. Return an array of error messages for fields that failed validation.

What's Next We have peered behind the curtain at Lattice's runtime introspection capabilities—type queries, tokenization, struct reflection, and dynamic evaluation. With the "Under the Hood" part complete, we are ready to put everything together. In Chapter 37, we will build a complete command-line tool from scratch, applying the language features, concurrency patterns, and standard library knowledge we have accumulated throughout this book.

Part XI

Real-World Lattice

Chapter 37

Building a CLI Tool

Every language earns its stripes when you build something real with it. So far we have explored Lattice's features one by one—phases, concurrency, pattern matching, modules. Now it is time to weave them together into a complete, polished command-line application: a file-search utility called `findi` that searches files for a pattern and prints matching lines with context.

Along the way we will use three libraries from the Lattice ecosystem: `cli` for argument parsing, `dotenv` for loading configuration from `.env` files, and `log` for structured, leveled logging. We will finish by reaching for two facilities the runtime itself provides—`exec_argv` for driving other programs, and `clat compile` plus the `clat-run` thin runtime for shipping the finished tool. By the end of this chapter you will have a template for every CLI tool you write in Lattice.

Read This Before You Type Along

The three libraries in this chapter live in the `lib/` directory of the Lattice repository, and at v0.6.4 two of them have rough edges that will bite you within thirty seconds of starting. `lib/cli.lat` does not parse at all until you make a one-character edit, and `lib/log.lat` loads on only one of the three backends. Both are spelled out precisely—with the exact error text—in Section 37.1.1 and Section 37.3.1. Read those two boxes first; everything else in the chapter assumes you have.

37.1 The `cli` Library for Argument Parsing

A command-line tool lives or dies by its interface. Users expect `--help` to work, short flags like `-v` to combine, and helpful error messages when they get something wrong. The `cli` library (found in `lib/cli.lat`) gives us all of this through a declarative API.

37.1.1 One Space Before You Begin

Try importing the library on a v0.6.4 build and nothing works at all:

Listing 37.1: The v0.6.4 import failure

```
$ clat findi.lat
vm error: import '/home/you/findi/lib/cli.lat': 65:21: parse error: unexpected token
    '||' in expression
164:5: parse error: unexpected token '}' in expression
391:5: parse error: unexpected token '}' in expression
394:1: parse error: unexpected token '}' in expression
```

Import failures report the *resolved* absolute path, so yours will name your own directory. The prefix varies by backend as well—`vm error:` on the default stack VM, plain `error:` under `--tree-walk`, `regvm error:` under `--regvm`—but the message after the prefix is identical on all three.

The library is older than the lexer rule that produced this message. Lattice spells an empty parameter list `| |`—with a space—because `||` is the logical-or operator, and the lexer commits to that reading before the parser ever gets a say. Two lines in `lib/cli.lat` still use the old spelling:

Listing 37.2: The two lines to change in lib/cli.lat

```
// line 65 and line 167, as shipped:
app.set("help", || {
app.set("parse", || {

// what v0.6.4 needs:
app.set("help", | | {
app.set("parse", | | {
```

Insert a space in each and the whole library works on every backend. The cascade of `unexpected token '}'` errors below the first one is the multi-error parser doing its job—it recovers and keeps going rather than stopping at the first failure—and all four disappear together.

This Is Your Own Empty-Closure Trap Too

The rule is not specific to `lib/cli.lat`. Every empty closure you write needs the space: `| | { ... }`, never `|| { ... }`. It is the single most common transcription error when porting older Lattice code forward, and the error message—`unexpected token '||' in expression`—points at the closure rather than at the concept, so it is worth memorizing now.

37.1.2 Creating a Parser

We begin by importing the library and creating a new application definition:

Listing 37.3: A minimal CLI parser

```
import "lib/cli" as cli

let app = cli.new("greet", "A friendly greeter")
app.flag("loud", "l", "Shout the greeting")
app.option("name", "n", "Who to greet", "World")

let opts = app.parse()
print(opts)
// Run with no arguments:
// {"name": World, "_args": [], "loud": false}
```

The `cli.new` function takes two arguments—the program name and a short description—and returns a Map packed with closures for configuring and executing the parser.

Notice the order of the keys in that output. A Lattice Map iterates in hash order, not insertion order, so the printed sequence will not match the order in which we registered things. Nothing downstream should depend on it.

CLI Application Map

The value returned by `cli.new` is a Map containing closures: `.flag(name, short, desc)` registers a boolean flag, `.option(name, short, desc, default)` registers a named option with a default value, `.arg(name, desc, required)` registers a positional argument, `.help()` generates help text, and `.parse()` parses the command-line arguments and returns a result Map.

37.1.3 Flags, Options, and Positional Arguments

The `cli` library distinguishes three kinds of command-line input:

- **Flags** are boolean switches. They default to `false` and become `true` when present. Think `--verbose` or `-v`.

- **Options** carry a value. They can appear as `--output file.txt`, `--output=file.txt`, or `-o file.txt`. Every option has a default.

- **Positional arguments** are the bare values that remain after flags and options are consumed. They can be required or optional.

Listing 37.4: Registering all three kinds of input

```
import "lib/cli" as cli

let app = cli.new("findi", "Search files for a pattern")

// Flags (boolean)
app.flag("verbose", "v", "Show extra information")
app.flag("count", "c", "Print only the count of matches")
app.flag("ignorecase", "i", "Case-insensitive search")

// Options (with default values)
app.option("output", "o", "Write results to a file", "")
app.option("context", "C", "Lines of context to show", "0")

// Positional arguments
app.arg("pattern", "The search pattern", true)     // required
app.arg("path", "File or directory to search", false) // optional
```

37.1.4 Parsing and the Result Map

Calling `app.parse()` reads the process arguments (via the built-in `args()` function), matches them against the registered flags, options, and positionals, and returns a result Map. Flags map to `Bool` values, options map to `String` values (or their defaults), and positional arguments map to `String` values. Any extra positional arguments land in the `"_args"` key as an array.

Listing 37.5: Working with parsed results

```
let opts = app.parse()

// Boolean flags
let is_verbose = opts.get("verbose")    // true or false

// String options (always strings---convert if needed)
let ctx_lines = to_int(opts.get("context"))

// Positional arguments
let pattern = opts.get("pattern")       // guaranteed present (required)
let path = opts.get("path")             // nil when omitted
```

An optional positional that the user leaves off really is `nil`, not an empty string, so test it with `path != nil` rather than `path != ""`. Anything beyond the registered positionals collects in `"_args"`: running `findi TODO src extra` leaves `"path"` as `"src"` and `"_args"` as `["extra"]`.

Options Are Always Strings

The `cli` library returns all option values as strings, including numeric defaults. When you register `app.option("port", "p", "Port number", "8080")`, the parsed value will be the string `"8080"`, not the integer `8080`. Use `to_int()` or `to_float()` to convert.

37.1.5 Automatic Help and Error Handling

The parser automatically intercepts `--help` and `-h`, printing a nicely formatted help message and exiting. You never need to register a help flag yourself.

Running our `findi` tool with `--help` produces:

Listing 37.6: Auto-generated help output

```
$ clat findi.lat --help
findi - Search files for a pattern

Usage: findi [OPTIONS] <pattern> [path]

Arguments:
  <pattern>   The search pattern (required)
  <path>      File or directory to search

Options:
  -v, --verbose    Show extra information
  -c, --count      Print only the count of matches
  -i, --ignorecase Case-insensitive search
  -o, --output     Write results to a file [default: ]
  -C, --context    Lines of context to show [default: 0]
  -h, --help       Show this help message
```

The `Usage:` line distinguishes required from optional—`<pattern>` versus `[path]`—but the `Arguments:` block below it renders every positional in angle brackets and marks the required ones with a trailing `(required)`. The option column is sized to the longest entry, so it grows as you register longer names.

If a user provides an unknown flag or forgets a required argument, the parser prints a one-line error, follows it with the full help text, and exits with status 1:

Listing 37.7: Argument errors

```
$ clat findi.lat
Error: missing required argument <pattern>

findi - Search files for a pattern
...

$ clat findi.lat --bogus TODO
Error: unknown option --bogus

findi - Search files for a pattern
...
```

Asking for `--help` exits with status 0, so `findi --help` in a shell script is not an error.

37.1.6 Combined Short Flags

Like many Unix tools, the `cli` library supports combining short flags into a single token. The invocation `findi -vic "TODO" src/` is equivalent to `findi -v -i -c "TODO" src/`. When combining, every character except the last must be a flag. The last character may be an option, in which case the next argument becomes its value:

Listing 37.8: Combined flags with a trailing option

```
# -v and -i are flags; -C is an option consuming "3"
findi -viC 3 "pattern" src/
```

Double-Dash Separator

The `cli` library supports the `--` separator. Everything after `--` is treated as a positional argument, even if it starts with a dash. This is essential for patterns like `findi -- -TODO- file.txt` where the pattern itself looks like a flag.

37.2 The `dotenv` Library for Configuration

CLI tools often need configuration that varies between environments—database URLs, API keys, default directories. Hardcoding these into source files is fragile. The `dotenv` library (in `lib/dotenv.lat`) loads key-value pairs from `.env` files into the process environment, so your code can read them with `env()`.

37.2.1 The Basics: `load` and `env`

Suppose our search tool needs a default search directory. We create a `.env` file alongside our script:

Listing 37.9: A `.env` file

```
# Default configuration for findi
FINDI_DEFAULT_PATH=./src
FINDI_MAX_RESULTS=100
FINDI_LOG_LEVEL=info
```

Then we load it at startup:

Listing 37.10: Loading a .env file

```
import "lib/dotenv" as dotenv

dotenv.load()  // loads .env from the current directory

let default_path = env("FINDI_DEFAULT_PATH")  // "./src"
let max_results = to_int(env("FINDI_MAX_RESULTS"))  // 100
```

The `dotenv.load()` function is deliberately forgiving: if no `.env` file exists, it silently does nothing. This is ideal for production, where configuration typically comes from the real environment rather than a file.

37.2.2 Loading Specific Files

When you need to load from a particular path—say, a per-environment configuration—use `dotenv.load_file`:

Listing 37.11: Loading a specific .env file

```
import "lib/dotenv" as dotenv

dotenv.load_file("config/.env.production")
```

Unlike `load()`, this function halts with an error if the file does not exist. It is meant for situations where the configuration file is mandatory:

Listing 37.12: A mandatory file that is not there

```
$ clat findi.lat
vm error: assertion failed: dotenv: file not found: config/.env.production
stack trace (most recent call last):
  [line 2] in <script>
  [line 349] in load_file()
```

The library raises this through an assertion, which is why the message carries the `assertion failed:` prefix on the two bytecode backends. Under `--tree-walk` you get the bare `error: dotenv: file not found: config/.env.production` instead.

37.2.3 Advanced Loading with Options

For more control, `dotenv.load_opts` accepts a Map of options:

Listing 37.13: Loading with options

```
import "lib/dotenv" as dotenv

let opts = Map::new()
opts.set("path", ".env.local")
opts.set("override", true)          // overwrite existing env vars
opts.set("required", ["DATABASE_URL", "SECRET_KEY"])

dotenv.load_opts(opts)
```

Override Behavior

By default, `dotenv` never overwrites environment variables that are already set. This means a real `DATABASE_URL` set in your shell takes precedence over whatever is in `.env`. Set `"override"` to `true` only when you explicitly want the file to win—typically during local development.

The `"required"` key is particularly useful for fail-fast behavior. If any listed variable is missing after loading, the program halts with a clear error message rather than silently using `nil` values:

Listing 37.14: A missing required variable

```
vm error: assertion failed: dotenv: required variable 'SECRET_KEY' is not set
```

The check runs once, at load time, which is exactly where you want a misconfigured deployment to fail—before the tool has opened a socket or written a byte.

37.2.4 Parsing Rules

The parser in `lib/dotenv.lat` handles a rich set of `.env` formats. Here is a file that exercises most of them:

Listing 37.15: A feature-rich .env file

```
# Comments start with #
BASIC=value

# Whitespace around = is trimmed
SPACED = hello

# Double-quoted values support escapes
GREETING="Hello,\nWorld!"

# Single-quoted values are literal (no escapes)
RAW='Hello,\nWorld!'

# Inline comments on unquoted values
PORT=3000 # default port

# export prefix is supported
export API_KEY=sk-abc123

# Variable expansion in double-quoted values
BASE_URL="https://api.example.com"
FULL_URL="${BASE_URL}/v2"

# Multiline double-quoted values
CERTIFICATE="-----BEGIN CERT-----
MIIB...
-----END CERT-----"
```

Feeding that file to `dotenv.parse` and printing each value shows every rule at work:

Listing 37.16: What the parser produces

```
BASIC       => [value]
SPACED      => [hello]
GREETING    => [Hello,
World!]
RAW         => [Hello,\nWorld!]
PORT        => [3000]
API_KEY     => [sk-abc123]
BASE_URL    => [https://api.example.com]
FULL_URL    => [https://api.example.com/v2]
CERTIFICATE => [-----BEGIN CERT-----
MIIB...
-----END CERT-----]
```

The double-quoted `GREETING` turned its `\n` into a real newline; the single-quoted `RAW` kept the two characters verbatim. `PORT` dropped its inline comment, `API_KEY` dropped its `export`, and `FULL_URL` expanded the earlier key. All three backends agree, character for character.

Parsing Without Setting

Sometimes you want to inspect a `.env` file without modifying the environment. Use `dotenv.parse(".env")` to get a Map of key-value pairs, or `dotenv.parse_string(content)` to parse an in-memory string.

37.3 The `log` Library for Structured Logging

A `print` statement is fine for quick debugging, but production CLI tools need leveled, timestamped, and optionally structured logging. The `log` library (in `lib/log.lat`) provides exactly that.

37.3.1 One Backend Only, For Now

At v0.6.4, `lib/log.lat` imports cleanly on exactly one of the three backends. Try it on the default stack VM and you do not get a diagnostic at all—the process aborts on the import with a segmentation fault, printing nothing on stdout or stderr:

Listing 37.17: lib/log on the default stack VM

```
$ clat findi.lat "TODO" ./src
$ echo $?
139
```

The silence is the whole problem: there is no message to search for, so the failure looks like your script rather than the import. The trigger is line 272 of the library, `_default_config.set("level", "info")`—a method call, at the top level of a module, on a binding the line above it declared. That combination is what breaks. Reduced to two tiny files it stops crashing and starts explaining itself:

Listing 37.18: A minimal reproduction (sub/m.lat)

```
eprint("module line 1")
let C = Map::new()
eprint("module line 3")
C.set("a", 1)
eprint("module line 5")
```

Listing 37.19: The importing script (main.lat)

```
eprint("script line 1")
import "sub/m" as m
eprint("script line 3")
```

Listing 37.20: Run under the default stack VM

```
$ clat main.lat
script line 1
module line 1
module line 3
script line 3
vm error: undefined variable 'C'
stack trace (most recent call last):
  [line 3] in <script>
  [line 4] in <closure>
```

Read the ordering carefully. `script line 3` runs *before* the module's fourth line, and by the time `C.set("a", 1)` finally executes, the module scope that held `C` is gone. Under `--tree-walk` and `--regvm` the same program prints all five lines in source order. The tree-walking interpreter has its own trouble with the real library—it crashes during the import—so `--regvm` is the backend to use for everything in this section:

Listing 37.21: Running a program that uses lib/log

```
$ clat --regvm findi.lat "TODO" ./src
```

Writing Modules That Dodge This

Your own modules can sidestep the problem entirely: keep module top level to `fn` declarations and plain `let` bindings, and move any method call into a function the importer calls. `let CONFIG = Map::new()` at module top level is fine on every backend; `CONFIG.set("k", 1)` on the next line is not. That is also better module design—an import that does work is an import that can surprise you.

37.3.2 Quick Start: Module-Level Functions

The fastest way to start logging is with the module-level convenience functions:

Listing 37.22: Module-level logging

```
import "lib/log" as log

log.info("Server started on port 8080")
log.warn("Connection pool running low")
log.error("Failed to read configuration file")
log.debug("Cache hit ratio: 0.87")  // suppressed at default level
```

These write to `stderr` using the default logger, which has a minimum level of `info`. That means `log.debug` messages are silenced unless you create a custom logger.

The output is human-readable, timestamped text:

Listing 37.23: Default log output format

```
$ clat --regvm log_demo.lat
2026-03-14 09:12:07 [INFO] Server started on port 8080
2026-03-14 09:12:07 [WARN] Connection pool running low
2026-03-14 09:12:07 [ERROR] Failed to read configuration file
```

Three lines, not four: the `log.debug` call produced nothing, because the default logger's threshold is `info`. Timestamps use `%Y-%m-%d %H:%M:%S` in local time, so yours will read differently. And every one of these lines went to *stderr*, which is what lets `findi --count . . . > counts.txt` capture results without the log getting mixed in.

37.3.3 Log Levels

The library defines four levels in increasing order of severity:

Level	Value	Purpose
`debug`	0	Detailed diagnostic information for development
`info`	1	Normal operational messages
`warn`	2	Something unexpected happened, but the tool can continue
`error`	3	Something failed; the operation cannot proceed normally

A logger only emits messages at or above its configured minimum level. Set the level to `"debug"` during development and `"warn"` or `"error"` in quiet production modes.

37.3.4 Creating a Custom Logger

For full control, create a logger with `log.new`:

Listing 37.24: Creating a custom logger

```
import "lib/log" as log

let cfg = Map::new()
cfg.set("level", "debug")           // show everything
cfg.set("file", "findi.log")        // also write to a file
cfg.set("format", "json")           // structured JSON output
cfg.set("prefix", "[findi]")        // prefix every message

let logger = log.new(cfg)

logger.info("Starting search")
logger.debug("Scanning directory: ./src")
```

The configuration Map supports four keys:

- `"level"` — minimum level: `"debug"`, `"info"`, `"warn"`, or `"error"` (default: `"info"`).
- `"file"` — path to a log file. Messages are appended to this file *in addition to* stderr (default: none).
- `"format"` — `"text"` for human-readable lines, `"json"` for machine-readable JSON (default: `"text"`).
- `"prefix"` — a string prepended to every message (default: `""`).

37.3.5 Structured Context

Each logging method accepts an optional second argument: a Map of contextual data. In text mode, the context is serialized as JSON and appended to the line. In JSON mode, the context keys are merged into the top-level log entry:

Listing 37.25: Logging with context

```
let ctx = Map::new()
ctx.set("path", "/api/users")
ctx.set("status", 404)
ctx.set("elapsed_ms", 12)

logger.warn("Route not found", ctx)
```

In text format, this produces:

Listing 37.26: Text log with context

```
2026-03-14 09:12:09 [WARN] [findi] Route not found
    {"elapsed_ms":12,"path":"/api/users","status":404}
```

In JSON format:

Listing 37.27: JSON log with context

```
{"elapsed_ms":12,"timestamp":"2026-03-14T09:12:09","message":"Route not
    found","path":"/api/users","prefix":"[findi]","level":"WARN","status":404}
```

Do Not Grep by Column

Both forms serialize a Map, and a Map iterates in hash order. The context keys did not come out in the order we set them, and the JSON entry's `"timestamp"` is not first. Parse these lines with a JSON reader, never with `cut -d, -f1`—the field order is not part of the contract and can change when a key is added.

Changing Levels at Runtime

A logger's threshold is fixed when the logger is built. `log.new` reads the level once, converts it to an Int, and each of the four emitting closures captures that Int—a scalar, copied into the closure—in its own `if 2 < min_level { return nil }` guard. Calling `logger.set("_level", 3)` afterwards updates the bookkeeping key that `set_level` reads, and nothing else—the closures keep testing the Int they already hold.

Use `log.set_level(logger, "error")` instead. It reads the stored configuration back out and calls `log.new` again, handing you a *new* logger with the same file, format, and prefix but a different threshold. The original keeps working at its old level.

Captured Containers Have Changed

Older Lattice guidance said flatly that closures capture everything by value, so no closure could ever mutate what it captured. That is no longer the whole story: since v0.6.4 a mutating builtin method on a captured *container* writes back to the captured cell, on all three backends.

```
flux acc = []
let add = |x| { acc.push(x) }
add(1)
add(2)
print(acc.len())        // 2
```

Scalars—Ints, Floats, Bools—are still copied into the closure, which is exactly why the logger's threshold cannot be changed after the fact. The distinction is worth internalizing: capture a container to share state, capture a number to freeze it.

37.4 Putting It All Together: A Complete CLI Application

Now let us build `findi`, our file-search tool. It reads configuration from `.env`, parses arguments with `cli`, and logs its activity with `log`. The tool searches a directory tree for lines matching a pattern and prints the results.

37.4.1 Project Layout

Listing 37.28: Project structure

```
findi/
  findi.lat          # main entry point
  .env               # default configuration
  lib/
    cli.lat          # copied from the Lattice repository
    dotenv.lat
    log.lat
```

Those three library files are copies, and they need to sit under the directory you *run from*, not next to the script. Import paths resolve relative to the working directory, so `clat findi/findi.lat` from one level up will not find `findi/lib/cli.lat`. Change into the project directory first. (And apply the one-space fix from Section 37.1.1 to your copy of `cli.lat`.)

And our `.env` file:

Listing 37.29: Default .env configuration

```
FINDI_DEFAULT_PATH=.
FINDI_LOG_LEVEL=info
FINDI_LOG_FORMAT=text
```

37.4.2 The Complete Source

Listing 37.30: findi.lat — A complete CLI file-search tool

```
import "lib/cli" as cli
import "lib/dotenv" as dotenv
import "lib/log" as log

// -- Load environment configuration ----------------------
dotenv.load()

// -- Set up argument parser ------------------------------
let app = cli.new("findi", "Search files for a text pattern")

app.flag("verbose", "v", "Show verbose output")
app.flag("count", "c", "Print only the match count")
app.flag("ignorecase", "i", "Case-insensitive matching")
app.option("output", "o", "Write results to a file", "")
app.option("context", "C", "Lines of context around matches", "0")
app.arg("pattern", "The text pattern to search for", true)
app.arg("path", "File or directory to search", false)

let opts = app.parse()

// -- Configure logging -----------------------------------
let log_cfg = Map::new()
let env_level = env("FINDI_LOG_LEVEL")
if typeof(env_level) == "String" {
    log_cfg.set("level", env_level)
} else {
    log_cfg.set("level", "info")
}
let env_format = env("FINDI_LOG_FORMAT")
if typeof(env_format) == "String" {
    log_cfg.set("format", env_format)
}
if opts.get("verbose") {
    log_cfg.set("level", "debug")
}
let logger = log.new(log_cfg)

// -- Extract parsed arguments ----------------------------
let pattern = opts.get("pattern")
let ignore_case = opts.get("ignorecase")
let count_only = opts.get("count")
let ctx_lines = to_int(opts.get("context"))
let output_file = opts.get("output")
```



```
// Determine search path: CLI arg > env default > "."
flux search_path = "."
```

37.4.3 Running the Tool

Let us take our tool for a spin against a two-file sample tree:

Listing 37.31: The tree we will search

```
src/main.lat                          src/utils.lat
------------------------------        ---------------------------------------
import "lib/cli" as cli               import "lib/log" as log
import "lib/dotenv" as dotenv         // TODO: another
// TODO: fix this                     fn helper(x: Int) -> Int { return x }
print("hi")
```

Listing 37.32: Running findi

```
$ clat --regvm findi.lat "import" ./src
2026-03-14 09:20:41 [INFO] Scanning 2 files
2026-03-14 09:20:41 [INFO] Search complete {"path":"./src","matches":3}
./src/utils.lat:1:import "lib/log" as log
./src/main.lat:1:import "lib/cli" as cli
./src/main.lat:2:import "lib/dotenv" as dotenv

$ clat --regvm findi.lat -c "TODO" ./src
2026-03-14 09:20:41 [INFO] Scanning 2 files
2026-03-14 09:20:41 [INFO] Search complete {"path":"./src","matches":2}
2

$ clat --regvm findi.lat -vi --context=1 "todo" ./src
2026-03-14 09:20:41 [DEBUG] Configuration loaded
2026-03-14 09:20:41 [DEBUG] Search parameters
    {"path":"./src","ignore_case":true,"pattern":"todo"}
2026-03-14 09:20:41 [INFO] Scanning 2 files
2026-03-14 09:20:41 [INFO] Search complete {"path":"./src","matches":2}
./src/utils.lat-import "lib/log" as log
./src/utils.lat:2:// TODO: another
./src/utils.lat-fn helper(x: Int) -> Int { return x }
./src/main.lat-import "lib/dotenv" as dotenv
./src/main.lat:3:// TODO: fix this
./src/main.lat-print("hi")
```

Three things to notice. The bracketed lines are the logger writing to stderr, interleaved here for readability—redirect stdout and they separate cleanly. The lowercase `todo` still found the uppercase `TODO` because `-i` was folded into `-vi`. And the file order follows `list_dir`, which is directory order, not alphabetical: if you want sorted output, sort the array `collect_files` returns.

37.4.4 Design Walkthrough

Let us highlight a few design choices worth remembering:

1. **Configuration cascade.** The search path has three sources: the CLI argument (highest priority), the `FINDI_DEFAULT_PATH` environment variable, and the hardcoded default `"."`. This cascade—command line overrides environment overrides defaults—is a pattern you will use in nearly every CLI tool.

2. **Verbose flag controls log level.** Rather than building a custom verbosity system, we lower the log level to `"debug"` when `--verbose` is passed. All those `logger.debug()` calls are free in normal operation.

3. **Fail fast with `dotenv.load_opts`.** For a tool that requires certain configuration (like a database URL), swap `dotenv.load()` for `dotenv.load_opts` with a `"required"` list. The tool will refuse to start if anything is missing, with a clear error message.

4. **Separate search logic from I/O.** The `search_file` function returns data (an array of match Maps), while the output section at the bottom handles formatting and writing. This separation makes it straightforward to add new output formats later.

5. **Context travels with the match.** Rather than re-reading the file when it is time to print, `search_file` slices the surrounding lines into `"before"` and `"after"` arrays inside each hit Map. With `--context 0` both arrays come back empty and the output collapses to one line per match, so there is no special case to write.

6. **The hit Map is built last.** We collect `before` and `after` into their own local arrays, then assemble the Map. Building the Map first and pushing into `hit.get("before")` would push into a copy.

Indexing Mutates, get Does Not

Since vo.6.4 a mutating method reached through an *index* writes back into the container. A mutating method reached through `.get()` does not—`get` returns a value, and you are mutating that value.

```
flux m = Map::new()
m.set("before", [])

m["before"].push(1)
print(len(m.get("before")))       // 1  -- wrote back

let arr = m.get("before")
arr.push(2)
print(len(m.get("before")))       // 1  -- arr was a copy
```

All three backends agree. The rule is worth stating out loud because the two spellings read as synonyms and are not: index to mutate, `get` to inspect.

Why Not Use `args()` Directly?

You could parse `args()` yourself with string manipulation, and for a two-flag script that might be fine. But the `cli` library handles edge cases you will eventually hit: combined short flags, `--` separators, missing required arguments, `--help` generation, and `--name=value` syntax. Writing all of that by hand is error-prone and tedious. Let the library do the work.

37.5 Driving Other Programs with exec_argv

Sooner or later a CLI tool needs to call another CLI tool. Lattice vo.6.4 added `exec_argv` for exactly this, and it is the one you should reach for:

Listing 37.33: Running a program and reading its output

```
let r = exec_argv("printf", ["%s", "hi"], nil)
print(r.get("stdout"))       // hi
print(r.get("exit_code"))    // 0
print(r.get("stderr"))       // (empty string)
```

The signature is `exec_argv(program, args, stdin, options = nil)`. There is no shell anywhere in that sentence—the arguments go straight into the `execvp` argument vector—so a filename containing

a space, a semicolon, or `$(rm -rf /)` remains nothing more than a filename. The `args` array excludes `argv[0]`; the runtime supplies it. `stdin` is a String to feed the child, or `nil` for none.

Listing 37.34: Feeding stdin and reading a non-zero exit

```
let hits = exec_argv("grep", ["-c", "TODO"], "a TODO\nb\nc TODO\n")
print(hits.get("stdout"))      // 2
print(hits.get("exit_code"))   // 0

let none = exec_argv("grep", ["-c", "ZZZ"], "a\n")
print(none.get("exit_code"))   // 1
print(none.get("stdout"))      // 0
```

A Non-Zero Exit Is Data, Not an Error

`grep` exiting 1 to mean "no matches" does not raise anything. You get a Map back and you decide what the number means. This is deliberate: exit codes are part of a program's interface, and a runtime that turned every one of them into an exception would make ordinary Unix plumbing painful.

The optional fourth argument is a Map of guards:

Listing 37.35: Bounding a child process

```
let opts = Map::new()
opts.set("timeout_ms", 200)

let out = try {
    exec_argv("sleep", ["5"], nil, opts)
} catch e {
    "caught: " + to_string(e.message)
}
print(out)   // caught: exec_argv: timed out after 200 ms
```

Alongside `"timeout_ms"` you can set `"max_stdout_bytes"` and `"max_stderr_bytes"`. Each has its own message:

Listing 37.36: Output limits

```
exec_argv: stdout exceeded max_stdout_bytes (8 bytes)
exec_argv: stderr exceeded max_stderr_bytes (4 bytes)
```

Blowing a limit raises a catchable error and the child's whole descendant tree is killed—a runaway subprocess cannot outlive the call that started it. A child that dies of `SIGPIPE` cannot take the interpreter down with it. All three backends produce identical messages.

Not Available in the Browser

A wasm32 build has no processes to exec. The browser path in `src/process_ops.c` raises a catchable error rather than returning a fake result—look for the `exec_argv: not available in browser` message. Wrap the call in `try` if your program is meant to run both natively and in a browser build.

37.6 Shipping the Tool

A finished tool should not require its users to have a compiler. `clat compile` turns a program into bytecode, and `clat-run`—a separate build target—executes that bytecode without carrying a lexer, parser, compiler, formatter, language server, or debugger:

Listing 37.37: Compiling and running bytecode

```
$ clat compile hello.lat -o hello.latc
$ clat-run hello.latc a b
Hello, Lattice!
[hello.latc, a, b]

$ clat compile --regvm hello.lat -o hello.rlat
$ clat-run hello.rlat x
Hello, Lattice!
[hello.rlat, x]
```

`clat-run` sniffs the file's magic bytes—`LATC` or `RLAT`—and picks the matching backend, so one binary serves both formats. Note that `args()` still reports the bytecode path as element zero, exactly as it reports the script path when you run source. Invoked with no arguments it identifies itself:

Listing 37.38: clat-run with no arguments

```
$ clat-run
Lattice bytecode runtime v0.6.4
Usage: clat-run <file.latc|file.rlat> [args...]
```

Hand it a `.lat` source file and it declines—this runtime genuinely cannot read source:

Listing 37.39: Source is not accepted

```
$ clat-run hello.lat
error: expected a .latc or .rlat bytecode file, got 'hello.lat'
```

How much smaller is it? Here are the v0.6.4 release artifacts, measured:

`clat-run` artifact	Size (bytes)
`darwin-aarch64`, `darwin-x86_64`	671,056
`freebsd-x86_64`	750,224
`openbsd-x86_64`	819,576
`netbsd-x86_64`	944,872
`windows-x86_64.exe`	999,424
`linux-riscv64`	3,921,952
`linux-aarch64`	4,352,808
`linux-x86_64`	5,335,312

Roughly two-thirds of a megabyte on the BSDs and macOS, about one megabyte on Windows, and several megabytes on the statically linked Linux builds. Quote the number for *your* target, not an average—the spread is more than seven to one.

`clat-run` Cannot Follow an `import`

This is the catch that will find you, and it finds you at the worst moment—after the build, on the target machine. An `import` of a `.lat` file is resolved at *run* time by lexing and parsing that file. The thin runtime has no lexer, so it aborts:

Listing 37.40: A compiled program that imports source

```
$ clat compile --regvm findi.lat -o findi.rlat
$ clat findi.rlat -c "TODO" ./src
2026-03-14 09:31:02 [INFO] Scanning 2 files
2026-03-14 09:31:02 [INFO] Search complete {"path":"./src","matches":2}
2

$ clat-run findi.rlat -c "TODO" ./src
fatal: lexer_new called in thin runtime (not supported)
```

The bytecode file compiles fine and runs fine under `clat`; only `clat-run` refuses. Our `findi` imports three libraries, so `clat-run` is not an option for it as written. Ship single-file programs through `clat-run`, or ship `clat` itself.

Chapter 34 covers the `.latc` and `.rlat` formats in detail—including the load-time verification that both runtimes perform on every chunk they read.

Exercises

1. **Add a `--max` option** to `findi` that limits the number of results printed. Default it to `0` (unlimited). Stop searching early once the limit is reached.

2. **Build a `wordcount` CLI tool** that counts words, lines, and characters in a file (like `wc`). Accept a `--lines`, `--words`, and `--chars` flag to control which counts are shown. Use the `cli` library for argument parsing and `log` for debug output.

3. **Environment file chain.** Modify the `findi` startup to load `.env` first, then `.env.local` with `override` set to `true`. This lets developers have personal overrides that are not committed to version control.

4. **JSON output mode.** Add a `--json` flag to `findi` that outputs results as a JSON array of objects, each with `"file"`, `"line"`, and `"text"` keys. Use `json_stringify` for the serialization.

5. **Custom log format.** Experiment with the `log` library's JSON format. Create a logger that writes JSON to a file and text to stderr. (Hint: you will need two loggers.)

6. **Sorted, stable output.** `collect_files` returns paths in directory order. Sort them before searching so two runs over the same tree always print matches in the same sequence.
7. **Shell out for the file list.** Replace `collect_files` with a call to `exec_argv("find", [path, "-type", "f"], nil)` and split the result on newlines. Then compare: which version handles a directory named `my files; rm -rf ~` correctly, and why? Add a `"timeout_ms"` guard so a search of an enormous tree cannot hang the tool.
8. **Make it shippable.** Inline the three imports—copy the handful of functions `findi` actually uses directly into `findi.lat`—then compile it and confirm it runs under `clat-run`. Note how much of each library you did not need.

What's Next

We have built a real command-line tool from scratch, using Lattice's `cli`, `dotenv`, and `log` libraries to handle the plumbing that every serious tool needs—and we have learned where those libraries are still catching up with the language, which is its own kind of useful. In the next chapter, we shift from the terminal to the network. We will build a complete web service—with HTTP routing, HTML templates, a database, and JSON APIs—using Lattice's `http_server`, `template`, and `orm` libraries. If you thought building a CLI was satisfying, wait until you see your first Lattice web page load in a browser.

Chapter 38

Building a Web Service

In the previous chapter we built a command-line tool. Now we are going to build something that talks to the world: a web service. By the end of this chapter we will have a running application with HTML pages, a JSON API, database persistence, and middleware—all written in Lattice.

We will use four pieces of the Lattice ecosystem: the `http_server` library for routing and request handling, the `template` library for rendering HTML, the `orm` library for SQLite persistence, and the built-in `json_parse`/`json_stringify` functions for JSON APIs. Let us start at the front door.

Which Backend to Run On

Everything in the first two sections—`http_server` and `template`—runs identically on all three backends. The `orm` library does not: at v0.6.4 it works under `--regvm` only, for the reason spelled out in Section 38.3.1. Because our finished application uses the ORM, every command in this chapter that starts a server is written `clat --regvm app.lat`.

38.1 The `http_server` Library

The `http_server` library (in `lib/http_server.lat`) builds on Lattice's raw TCP networking primitives—`tcp_listen`, `tcp_accept`, `tcp_read`, `tcp_write`, and `tcp_close`—to provide a high-level, Express-style API for building HTTP applications.

38.1.1 Hello, Web

The smallest possible web server fits in a handful of lines:

Listing 38.1: A minimal Lattice web server

```
import "lib/http_server" as http

let app = http.new()

app.get("/", |req, res| {
    return http.response(200, "Hello from Lattice!")
})

app.listen(8080)
```

Run this, open a browser to `http://localhost:8080`, and you will see the greeting. On the command line the whole exchange looks like this:

Listing 38.2: The smallest server, answering

```
$ clat hello_web.lat
Lattice HTTP server listening on http://localhost:8080

$ curl -i http://localhost:8080/
HTTP/1.1 200 OK
Content-Length: 19
Connection: close
Content-Type: text/plain

Hello from Lattice!
```

Note `Connection: close`: the server answers one request per connection and hangs up. There is no keep-alive, no connection pool, and no thread pool—requests are served one at a time in a single loop.

Let us break down what happens:

1. `http.new()` creates an application Map containing route tables and a middleware stack.
2. `app.get("/", handler)` registers a handler for GET requests to `/`.
3. The handler receives a request Map (`req`) and a response context Map (`res`), and returns a response Map built by `http.response`.

4. `app.listen(8080)` opens a TCP socket and enters a loop that accepts connections, parses HTTP requests, runs the middleware pipeline, dispatches to the matching handler, and sends back the formatted response.

The Request Map

Every handler receives a `req` Map with these keys:

- `"method"` — the HTTP method (`"GET"`, `"POST"`, etc.)
- `"path"` — the URL path without query string
- `"raw_path"` — the full URL including query string
- `"query"` — a Map of parsed query parameters
- `"headers"` — a Map of HTTP headers
- `"body"` — the raw request body as a string
- `"cookies"` — a Map of parsed cookie name-value pairs
- `"peer"` — the client's address

38.1.2 Response Helpers

The library provides several functions for building response Maps:

Listing 38.3: Response helper functions

```
import "lib/http_server" as http

// Plain text (Content-Type: text/plain)
let text_res = http.response(200, "OK")

// JSON (Content-Type: application/json)
let data = Map::new()
data.set("status", "healthy")
data.set("uptime", 3600)
let json_res = http.json(200, data)

// HTML (Content-Type: text/html)
let html_res = http.html(200, "<h1>Welcome</h1>")

// Redirect (302 by default)
let redir = http.redirect("/dashboard")

// Redirect with custom status
let perm_redir = http.redirect("/new-path", 301)
```

Each helper returns a Map with `"status"`, `"body"`, `"headers"`, and `"_cookies"` keys. The server serializes this into a proper HTTP response before sending it over the wire.

38.1.3 Route Registration

The application object provides registration closures for all standard HTTP methods:

Listing 38.4: Registering routes for different HTTP methods

```
import "lib/http_server" as http

let app = http.new()

app.get("/items", |req, res| {
    return http.json(200, get_all_items())
})

app.post("/items", |req, res| {
    let body = req.get("body")
    let item = json_parse(body)
    return http.json(201, create_item(item))
})

app.put("/items/update", |req, res| {
    let body = req.get("body")
    let item = json_parse(body)
    return http.json(200, update_item(item))
})

app.delete("/items/remove", |req, res| {
    return http.json(200, remove_item(req))
})

// Register for ALL methods at once
app.all("/health", |req, res| {
    return http.response(200, "OK")
})
```

The available methods are `app.get`, `app.post`, `app.put`, `app.delete`, `app.patch`, `app.options`, and `app.all`.

Dispatch is a linear scan that matches method *and* path together, and anything that falls off the end of the table gets a 404:

Listing 38.5: Unmatched requests

```
$ curl http://localhost:8080/nope
404 Not Found: /nope

$ curl -X POST http://localhost:8080/items/update    # registered for PUT
404 Not Found: /items/update
```

There Is No 405

A request for a path that exists under a *different* method gets 404 Not Found, not 405 Method Not Allowed. `lib/http_server.lat` does define a `_method_not_allowed_response` helper, but as of v0.6.4 the dispatcher never calls it—the route scan compares method and path in a single condition and has no notion of "right path, wrong method."
If your API's clients rely on the distinction, register a catch-all with `app.all` on each path and branch on `req.get("method")` yourself.

38.1.4 Query Parameters

Query strings are automatically parsed into the `"query"` Map on the request:

Listing 38.6: Reading query parameters

```
// Request: GET /search?q=lattice&page=2
app.get("/search", |req, res| {
    let query = req.get("query")
    let search_term = query.get("q")      // "lattice"
    flux page = 1
    if query.has("page") {
        page = to_int(query.get("page"))  // 2
    }
    // ... perform search ...
    return http.json(200, results)
})
```

The `query.has` check is not decoration. A missing key yields `nil`, and `to_int(nil)` raises `to_int(): cannot convert this type to Int`. An uncaught error inside a handler becomes a 500 for that request—the server prints the error and keeps serving—but a 500 is not the answer you wanted to give someone who typed a URL by hand.

Query Values Arrive Percent-Encoded

The query parser splits on `&` and `=` and stops there. It does no percent-decoding, so `?y=hello%20world` gives you the nine-character string `hello%20world`, and a `+` stays a `+`. Decode it yourself with the built-in `url_decode`, which handles both forms:

```
print(url_decode("hello%20world"))   // hello world
print(url_decode("a+b"))             // a b
print(url_encode("hello world&x=1")) // hello%20world%26x%3D1
```

Skipping this is the difference between a search box that finds "ice cream" and one that searches for `ice%20cream`.

38.1.5 Middleware

Middleware functions run before your route handlers, forming a pipeline that can inspect requests, modify responses, or short-circuit the chain entirely. Register middleware with `app.use`:

Listing 38.7: Using middleware

```
import "lib/http_server" as http

let app = http.new()

// Built-in request logger
app.use(http.logger())

// Built-in CORS middleware
app.use(http.cors())

// Built-in JSON body parser
app.use(http.body_parser())

// Built-in security headers
app.use(http.security_headers())

app.get("/", |req, res| {
    return http.response(200, "Hello!")
})

app.listen(3000)
```

Middleware Signature

A middleware is a function with the signature `|req, res, next|`. To continue the pipeline, call `next(req, res)` and return its result. To short-circuit (for example, to reject an unauthenticated request), return a response directly without calling `next`.

Here is a custom middleware that checks for an API key:

Listing 38.8: Writing custom middleware

```
fn require_api_key() -> Fn {
    return |req, res, next| {
        let headers = req.get("headers")
        flux api_key = ""
        if headers.has("X-API-Key") {
            api_key = headers.get("X-API-Key")
        }
        if api_key != "secret-key-123" {
            return http.json(401, "Unauthorized")
        }
        return next(req, res)
    }
}

app.use(require_api_key())
```

Listing 38.9: The gate in action

```
$ curl http://localhost:8080/
"Unauthorized"

$ curl -H "X-API-Key: secret-key-123" http://localhost:8080/
Hello!

$ curl -H "x-api-key: secret-key-123" http://localhost:8080/
"Unauthorized"
```

Header Lookups Are Case-Sensitive

That third request had a valid key and was still rejected. HTTP header names are case-insensitive by specification, but the request parser stores each name exactly as the client sent it, and Map lookup is an exact string match. A client that sends `x-api-key`—and plenty do—will miss a `headers.has("X-API-Key")` check entirely.

Normalize before you compare. Fold the header Map's keys to lowercase once, in a middleware near the front of the stack, and have every downstream handler look up the lowercase form.

The built-in middleware that ships with the library includes:

- `http.logger()` — logs method, path, status, and response time.
- `http.cors()` — handles CORS preflight requests and adds appropriate headers. Accepts optional configuration via `http.cors_opts()`.
- `http.body_parser()` — automatically parses JSON request bodies when the `Content-Type` is `application/json`, storing the result in `req.get("parsed_body")`.
- `http.security_headers()` — adds `X-Content-Type-Options`, `X-Frame-Options`, and other security headers.
- `http.compression()` — sets `Vary: Accept-Encoding` for correct proxy caching.

With `http.logger()` installed, each served request produces one line on stdout—method, path, status, elapsed time:

Listing 38.10: The access log

```
Lattice HTTP server listening on http://localhost:8080
GET /items 200 0ms
GET /nope 404 0ms
```

And with `http.security_headers()` and `http.cors()` both in the stack, every response picks up their headers:

Listing 38.11: What the middleware adds

```
$ curl -i http://localhost:8080/items
HTTP/1.1 200 OK
Content-Length: 34
Connection: close
Content-Type: application/json
X-Frame-Options: DENY
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Referrer-Policy: strict-origin-when-cross-origin
Access-Control-Allow-Origin: *

{"uptime":3600,"status":"healthy"}
```

A CORS preflight is answered by the middleware itself and never reaches a handler—`OPTIONS` returns `204 No Content` with `Access-Control-Allow-Methods`, `-Headers`, and `-Max-Age` set.

38.1.6 Cookies

The library provides helpers for reading and setting cookies:

Listing 38.12: Cookie handling

```
app.get("/profile", |req, res| {
    // Read cookies from the request
    let cookies = req.get("cookies")
    flux username = "Guest"
    if cookies.has("username") {
        username = cookies.get("username")
    }

    let r = http.html(200, "<h1>Hello, " + username + "</h1>")

    // Set a cookie on the response
    let opts = http.cookie_opts()  // sensible defaults
    opts.set("max_age", 86400)     // 1 day
    return http.set_cookie(r, "visited", "true", opts)
})

app.get("/logout", |req, res| {
    let r = http.redirect("/")
    let opts = http.cookie_opts()
    return http.clear_cookie(r, "username", opts)
})
```

The `http.cookie_opts()` function returns a Map with sensible defaults: `Path=/`, `HttpOnly=true`, `SameSite=Lax`. You can override any of these before passing the Map to `set_cookie`. The two routes above put these on the wire:

Listing 38.13: The Set-Cookie headers produced

```
$ curl -i -b "username=Alice" http://localhost:8080/profile
HTTP/1.1 200 OK
Set-Cookie: visited=true; Path=/; Max-Age=86400; HttpOnly; SameSite=Lax
...
<h1>Hello, Alice</h1>

$ curl -i http://localhost:8080/logout
HTTP/1.1 302 Found
Set-Cookie: username=; Path=/; Max-Age=0; HttpOnly; SameSite=Lax
Location: /
```

Clearing a cookie is the same mechanism with `Max-Age=0` and an empty value—the browser does the forgetting.

Maps Passed to Functions Are Copies

A Map handed to a function is a copy, so `http.set_cookie` cannot reach back and modify your response. It returns a *new* response Map, and you must use the return value: `let r = http.set_cookie(r, ...)`. Forgetting this is a common source of "my cookies are not being set" bugs.

This is the one place where the v0.6.4 change to captured-container mutation does *not* help you. A closure that mutates a container it captured now writes back—but a function parameter is still a copy, and always was.

38.2 The `template` Library for HTML Templating

Building HTML strings by hand is tedious and error-prone. The `template` library (in `lib/template.lat`) provides a Mustache/Jinja2-inspired template engine with variable interpolation, conditionals, loops, filters, includes, and template inheritance.

38.2.1 Variables and Escaping

The core operation is variable interpolation with double-braces:

Listing 38.14: Template variable interpolation

```
import "lib/template" as tmpl

let t = tmpl.compile("Hello, {{name}}!")

let data = Map::new()
data.set("name", "World")

let result = tmpl.render(t, data)
// result: "Hello, World!"
```

By default, `{{name}}` HTML-escapes the output—so if `name` is `"<script>alert('xss')</script>"`, it becomes `"<script>..."`. When you need raw, unescaped output (for example, pre-rendered HTML), use triple-braces:

Listing 38.15: Raw vs. escaped output

```
let t = tmpl.compile("{{{html_content}}}")

let data = Map::new()
data.set("html_content", "<b>Bold</b>")

let result = tmpl.render(t, data)
// result: "<b>Bold</b>"  (not escaped)
```

Nested map access uses dot notation:

Listing 38.16: Dot notation for nested values

```
let t = tmpl.compile("{{user.name}} ({{user.email}})")

let user = Map::new()
user.set("name", "Alice")
user.set("email", "alice@example.com")

let data = Map::new()
data.set("user", user)

let result = tmpl.render(t, data)
// result: "Alice (alice@example.com)"
```

38.2.2 Filters

Filters transform a value inline using the pipe syntax:

Listing 38.17: Using template filters

```
let t = tmpl.compile("""
  Name: {{name | upper}}
  Bio: {{bio | truncate(50)}}
  Tags: {{tags | join(", ")}}
  Role: {{role | default("member")}}
""")
```

The available filters are:

Filter	Description
`upper`	Converts to uppercase
`lower`	Converts to lowercase
`capitalize`	Capitalizes the first letter
`title`	Title-cases each word
`trim`	Strips leading/trailing whitespace
`escape_html`	HTML-escapes the value
`length`	Returns the length
`string`	Converts to a string
`reverse`	Reverses the string
`default("val")`	Uses `val` if the variable is nil or empty
`truncate(n)`	Truncates to `n` characters, adding `...`
`replace(a, b)`	Replaces occurrences of `a` with `b` — see the warning below
`join(", ")`	Joins array elements with the given separator

Filters can be chained: `{{name | trim | upper}}` first trims, then uppercases.

The `replace` Filter Takes Unquoted Arguments

Every other two-argument filter is written with quotes, and `replace` looks like it should be too. It is not, and the failure is silent:

Listing 38.18: The quoted form does nothing

```
{{v | replace("a","b")}}   with v = "banana"  ->  banana
{{v | replace(a,b)}}       with v = "banana"  ->  bbnbnb
```

The argument parser strips one leading and one trailing quote from the whole argument text, then splits on the first comma. Given `"a","b"` it strips the outermost pair, leaving `a","b`, and hands `replace` the search string `a"`—which occurs nowhere—so the value comes through untouched. `join` and `default` take a single argument and are unaffected: `{{tags | join(", ")}}` works exactly as written, separator space and all.

38.2.3 Conditionals

The library supports both Mustache-style and Jinja2-style conditionals:

Listing 38.19: Mustache-style conditionals

```
let t = tmpl.compile("""
  {{#if logged_in}}
    Welcome back, {{username}}!
  {{else}}
    Please log in.
  {{/if}}
""")
```

Listing 38.20: Jinja2-style conditionals with elif

```
let t = tmpl.compile("""
  {% if is_admin %}
    <span class="badge">Admin</span>
  {% elif is_editor %}
    <span class="badge">Editor</span>
  {% else %}
    <span class="badge">User</span>
  {% endif %}
""")
```

The `{% elif %}` chain can be as long as you like, and `{% if %}` blocks nest. Mustache-style blocks have no `elif`; nest `{{#if}}` inside `{{else}}` if you need the same shape there.

Truthiness follows Lattice conventions: `nil`, `false`, `0`, `0.0`, and `""` are all falsy. Everything else is truthy.

The negated conditional `{{#unless ...}}` works like `{{#if ...}}` with inverted logic:

Listing 38.21: Using unless

```
let t = tmpl.compile("""
  {{#unless errors}}
    <p>No errors found.</p>
  {{/unless}}
""")
```

38.2.4 Loops

Iteration comes in two flavors. Mustache-style `each` works well for arrays of maps:

Listing 38.22: Mustache-style each loop

```
let t = tmpl.compile("""
  <ul>
  {{#each items}}
    <li>{{name}} - {{price}}</li>
  {{/each}}
  </ul>
""")
```

Inside an `each` block, the keys of each Map item are promoted into scope. The special variables `{{@index}}`, `{{@first}}`, and `{{@last}}` are also available.

Jinja2-style `for` loops give you a named loop variable:

Listing 38.23: Jinja2-style for loop

```
let t = tmpl.compile("""
  <table>
  {% for user in users %}
    <tr class="{% if loop.first %}first{% endif %}">
      <td>{{loop.index1}}</td>
      <td>{{user.name}}</td>
    </tr>
  {% endfor %}
  </table>
""")
```

The `loop` variable inside a Jinja2 for-loop provides: `loop.index` (0-based), `loop.index1` (1-based), `loop.first`, `loop.last`, and `loop.length`.

38.2.5 Includes and Template Inheritance

For larger applications, you can split templates into partials and base layouts:

Listing 38.24: Including a partial

```
// In your template:
// {% include "templates/header.html" %}
// ... page content ...
// {% include "templates/footer.html" %}
```

Template inheritance lets child templates override blocks defined in a parent:

Listing 38.25: base.html — a base layout

```
<html>
<head><title>{% block title %}My App{% endblock %}</title></head>
<body>
  <nav>...</nav>
  <main>{% block content %}{% endblock %}</main>
  <footer>...</footer>
</body>
</html>
```

Listing 38.26: page.html — a child template

```
{% extends "templates/base.html" %}
{% block title %}Dashboard{% endblock %}
{% block content %}
  <h1>Dashboard</h1>
  <p>Welcome back!</p>
{% endblock %}
```

When you compile and render `page.html`, the engine loads `base.html`, finds the block definitions in the child, and substitutes them. The result is a complete HTML page with "Dashboard" as the title and the dashboard content in the `<main>` element.

Listing 38.27: Rendering an inherited template

```
import "lib/template" as tmpl

let page = tmpl.from_file("templates/page.html")
let output = tmpl.render(page, Map::new())
```

Compile Once, Render Many

The `tmpl.compile` function parses a template string into a segment tree. If you are rendering the same template with different data (as you would in a web server), compile once at startup and call `tmpl.render` on each request. This avoids re-parsing the template on every request.

38.3 The `orm` Library for SQLite Persistence

Most web services need to persist data. The `orm` library (in `lib/orm.lat`) provides a lightweight ORM layer over Lattice's SQLite native extension. It loads the extension via `require_ext("sqlite")` and wraps the raw SQL operations in a model-based API.

38.3.1 Backend and Extension Prerequisites

Two things have to be true before any of this runs.

First, the SQLite extension has to be somewhere the loader trusts. As of v0.6.4, `require_ext` searches `~/.lattice/ext/` and `$LATTICE_EXT_PATH` first, and consults a working-directory `./extensions/` *only* when `LATTICE_EXT_ALLOW_CWD` is set. Copying `sqlite.dylib` (or `.so`) into `~/.lattice/ext/` is the arrangement that always works.

Second—and this is the one that will confuse you—the library runs on one backend. `lib/orm.lat` opens with `let _sqlite = require_ext("sqlite")` and then, on the very next line, calls `_sqlite.get("open")`. That is a method call on a module-level binding at a module's top level, and on the default stack VM it aborts the process outright—no error text, no stack trace, nothing on either stream:

Listing 38.28: lib/orm on the default stack VM

```
$ clat app.lat
$ echo $?
139
```

Because nothing is printed, the crash reads as a fault in your own first line rather than in the import. The tree-walking interpreter gets further—the import succeeds—but crashes inside `orm.model`. That leaves `--regvm`, where the whole library works end to end. The same defect keeps `lib/log.lat` off two backends; Section 37.3.1 reproduces it in two tiny files and explains how to keep your own modules clear of it.

Sanity-Check Your Setup First

Before debugging your schema, confirm the plumbing:

```
let s = require_ext("sqlite")
print("loaded")
let open = s.get("open")
print("got open")
```

That runs on every backend—`require_ext` itself is fine, and so is calling a method on a binding in your *own* script. It is only the same pattern inside an imported module that breaks.

38.3.2 Connecting and Defining Models

Listing 38.29: Setting up a database and model

```
import "lib/orm" as orm

// Connect to a file-based database (or ":memory:" for tests)
let db = orm.connect("app.db")

// Define the schema as a Map of column -> SQL type
let schema = Map::new()
schema.set("id", "INTEGER PRIMARY KEY AUTOINCREMENT")
schema.set("title", "TEXT NOT NULL")
schema.set("body", "TEXT")
schema.set("created_at", "TEXT")

// Create a model bound to the "posts" table
let Post = orm.model(db, "posts", schema)

// Create the table (safe to call multiple times)
let create_table = Post.get("create_table")
create_table(0)
```

The Dummy Argument

You may notice that `create_table(0)` passes a dummy argument. This is because the ORM's closure signatures in `lib/orm.lat` accept a parameter (e.g., `|_x|`) even when one is not semantically needed. Pass any value—`0`, `nil`, or `""`—to satisfy the call.

38.3.3 CRUD Operations

The model Map provides closures for all the standard operations:

Listing 38.30: Creating records

```
let create = Post.get("create")

let post_data = Map::new()
post_data.set("title", "Hello World")
post_data.set("body", "This is my first post.")
post_data.set("created_at", time_format(time(), "%Y-%m-%d %H:%M:%S"))

let post_id = create(post_data)
print("Created post with id: " + to_string(post_id))
```

Listing 38.31: Reading records

```
// Find by id
let find = Post.get("find")
let post = find(post_id)
print(post.get("title"))  // "Hello World"

// Get all records
let all = Post.get("all")
let posts = all(0)
for p in posts {
    print(p.get("title"))
}

// Query with a WHERE clause
let where_fn = Post.get("where")
let recent = where_fn("created_at > ?", ["2025-01-01"])
```

Listing 38.32: Updating and deleting records

```
// Update
let update = Post.get("update")
let changes = Map::new()
changes.set("title", "Hello World (Updated)")
update(post_id, changes)

// Delete
let delete_fn = Post.get("delete")
delete_fn(post_id)

// Count
let count = Post.get("count")
print("Total posts: " + to_string(count(0)))
```

Running all of that against a fresh `app.db` produces:

Listing 38.33: The CRUD sequence, executed

```
$ clat --regvm posts_demo.lat
Created post with id: 1
Hello World
all: Hello World
where count: 1
after update: Hello World (Updated)
Total posts: 1
after delete count: 0
```

`create` returns the new row's integer id, `find` returns a Map (or `nil`), and `all`, `where` return Arrays of Maps.

The `where` closure accepts a SQL condition string with `?` placeholders and an array of parameter values. The parameters are safely bound by the SQLite extension, so you get protection against SQL injection for free.

Listing 38.34: Closing the database

```
// Always close when done
orm.close(db)
```

Always Close Your Database

SQLite databases should be closed when your application shuts down to ensure all writes are flushed. In a long-running web server you typically keep the connection open for the lifetime of the process, but make sure to call `orm.close(db)` if you ever exit gracefully.

38.4 JSON APIs with `json_parse` and `json_stringify`

Lattice's JSON support is implemented as built-in functions backed by a C parser and serializer (in `src/json.c`). The recursive-descent parser handles the full JSON specification, including Unicode escapes, nested structures, and all JSON types.

38.4.1 Parsing JSON

`json_parse` takes a JSON string and returns the corresponding Lattice value—a Map for objects, an Array for arrays, and the appropriate scalar types for strings, numbers, booleans, and null:

Listing 38.35: Parsing JSON strings

```
let data = json_parse("{\"name\":\"Alice\",\"age\":30}")
// data is a Map
print(data.get("name"))  // "Alice"
print(data.get("age"))   // 30

let items = json_parse("[1, 2, 3]")
// items is an Array
print(items[0])  // 1

let nested = json_parse("""
    {
        "users": [
            {"name": "Alice", "active": true},
            {"name": "Bob", "active": false}
        ],
        "total": 2
    }
""")
let users = nested.get("users")
print(users[0].get("name"))  // "Alice"
```

JSON null maps to Lattice `nil`. JSON booleans map to Lattice `true` and `false`. JSON integers stay as `Int`, and numbers with decimal points or exponents become `Float`—`json_parse("1e3")` is a Float, not an Int.

The Parser Got Strict in v0.6.4

Two inputs that older Lattice accepted are now errors, and both matter for a service taking JSON from the network:

Listing 38.36: Duplicate keys and oversized integers

```
try { json_parse("{\"a\":1,\"a\":2}") } catch e { print(e.message) }
// json_parse error at position 10: duplicate object key

try { json_parse("99999999999999999999999999") } catch e { print(e.message) }
// json_parse error at position 26: integer out of signed 64-bit range
```

There is no more last-key-wins, and no more silent integer wraparound. Errors carry the byte position, input must be valid shortest-form UTF-8, and `\uXXXX` surrogate pairs decode correctly. All three backends produce identical messages.

The practical consequence: a request body that used to parse into *something* may now raise. Wrap `json_parse` in `try` at every network boundary and answer 400, which is exactly what `http.body_parser()` does on your behalf.

38.4.2 Serializing to JSON

`json_stringify` converts a Lattice value to a JSON string:

Listing 38.37: Serializing Lattice values to JSON

```
let data = Map::new()
data.set("name", "Alice")
data.set("scores", [95, 87, 92])
data.set("active", true)

let json_text = json_stringify(data)
print(json_text)
// {"scores":[95,87,92],"name":"Alice","active":true}
```

Read that output next to the code that produced it. The keys came back in a different order than we set them, because a Map iterates in hash order. Any consumer of your API that cares about key order is already broken; any test of yours that string-compares JSON output is about to be.

The serializer handles Maps, Arrays, Tuples, Buffers, strings, numbers, booleans, nil, and Refs. Tuples and Buffers both serialize as arrays—`(1, "a", true)` becomes `[1,"a",true]`, and a three-byte Buffer holding 65 becomes `[65,0,0]`—and a Ref serializes as whatever it points at. Types with no JSON representation—closures, channels, ranges, structs—raise a catchable error:

Listing 38.38: What cannot be serialized

```
struct Point { x: Int }

let out = try {
    json_stringify(Point { x: 1 })
} catch e {
    "caught: " + to_string(e.message)
}
print(out)   // caught: json_stringify: unsupported value type
```

Round-Tripping

`json_parse(json_stringify(value))` will faithfully round-trip any value composed of Maps, Arrays, strings, numbers, booleans, and nil. This makes JSON a convenient serialization format for inter-process communication, configuration files, and API payloads.

38.4.3 JSON in HTTP Handlers

Combining JSON with the HTTP server is where things get practical. Here is a pattern you will use constantly:

Listing 38.39: A JSON API endpoint

```
import "lib/http_server" as http

let app = http.new()
app.use(http.body_parser())

app.post("/api/echo", |req, res| {
    let parsed = req.get("parsed_body")
    if parsed == nil {
        return http.json(400, "Invalid JSON")
    }
    // Echo back the parsed data with a timestamp
    parsed.set("received_at", time_format(time(), "%Y-%m-%dT%H:%M:%S"))
    return http.json(200, parsed)
})
```

The `http.body_parser()` middleware automatically parses JSON request bodies and stores the result in `req.get("parsed_body")`, saving you from calling `json_parse` manually on every handler.

It parses only when the `Content-Type` header contains `application/json`, and it wraps the parse in `try`—so a malformed body leaves `"parsed_body"` as `nil` rather than blowing up the request. Both failure modes therefore look identical to your handler, which is why the `parsed == nil` check above is the whole guard you need:

Listing 38.40: Two ways to get nil

```
$ curl -X POST -d 'notjson' http://localhost:8080/api/echo
"Invalid JSON"

$ curl -X POST -H "Content-Type: application/json" -d '{bad}' \
    http://localhost:8080/api/echo
"Invalid JSON"
```

38.5 A Complete Web Application from Scratch

Let us bring everything together and build **Lattice Notes**—a note-taking web application with HTML pages for humans and a JSON API for machines.

38.5.1 Architecture

Listing 38.41: Project structure

```
lattice-notes/
  app.lat                # main entry point
  templates/
    base.html            # base layout
    index.html           # note listing page
    note.html            # single note page
  lib/                   # copied from the Lattice repository
    http_server.lat
    template.lat
    orm.lat
    dotenv.lat
    log.lat
```

Both the `import` paths and the `{% extends %}` paths inside the templates resolve relative to the working directory, so run `clat` from inside `lattice-notes/`. Starting it from one level up gives you `import: cannot find 'lib/http_server.lat'` before the first line executes.

The application will have these routes:

Method	Path	Description
GET	`/`	List all notes (HTML)
GET	`/notes/view`	View a single note (HTML, `?id=N`)
POST	`/notes/create`	Create a note from form data
GET	`/api/notes`	List all notes (JSON)
POST	`/api/notes`	Create a note (JSON)
GET	`/api/notes/one`	Get a note by id (JSON, `?id=N`)

38.5.2 Templates

First, our base layout:

Listing 38.42: templates/base.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>{% block title %}Lattice Notes{% endblock %}</title>
  <style>
    body { font-family: sans-serif; max-width: 800px;
           margin: 0 auto; padding: 20px; }
    .note { border: 1px solid #ddd; padding: 12px;
            margin: 8px 0; border-radius: 4px; }
    .note h3 { margin: 0 0 8px 0; }
    form input, form textarea { display: block;
            width: 100%; margin: 8px 0; padding: 8px; }
    form button { padding: 8px 16px; }
  </style>
</head>
<body>
  <h1><a href="/">Lattice Notes</a></h1>
  {% block content %}{% endblock %}
</body>
</html>
```

The index page lists all notes and includes a creation form:

Listing 38.43: templates/index.html

```
{% extends "templates/base.html" %}
{% block title %}All Notes{% endblock %}
{% block content %}
  <h2>All Notes ({{note_count}})</h2>

  {{#each notes}}
    <div class="note">
      <h3><a href="/notes/view?id={{id}}">{{title}}</a></h3>
      <p>{{body | truncate(120)}}</p>
      <small>{{created_at}}</small>
    </div>
  {{/each}}

  {{#unless notes}}
    <p>No notes yet. Create one below!</p>
  {{/unless}}

  <hr>
  <h2>New Note</h2>
  <form method="POST" action="/notes/create">
    <input name="title" placeholder="Title" required>
    <textarea name="body" placeholder="Body" rows="4"></textarea>
    <button type="submit">Create Note</button>
  </form>
{% endblock %}
```

And the single-note view:

Listing 38.44: templates/note.html

```
{% extends "templates/base.html" %}
{% block title %}{{title}}{% endblock %}
{% block content %}
  <h2>{{title}}</h2>
  <p>{{body}}</p>
  <small>Created: {{created_at}}</small>
  <p><a href="/">Back to all notes</a></p>
{% endblock %}
```

38.5.3 The Application Source

Listing 38.45: app.lat — Lattice Notes web application

```
import "lib/http_server" as http
import "lib/template" as tmpl
import "lib/orm" as orm
import "lib/dotenv" as dotenv
import "lib/log" as log

// -- Configuration ---------------------------------------
dotenv.load()

let log_cfg = Map::new()
log_cfg.set("level", "info")
let logger = log.new(log_cfg)

// -- Database setup --------------------------------------
let db = orm.connect("notes.db")

let schema = Map::new()
schema.set("id", "INTEGER PRIMARY KEY AUTOINCREMENT")
schema.set("title", "TEXT NOT NULL")
schema.set("body", "TEXT")
schema.set("created_at", "TEXT")

let Note = orm.model(db, "notes", schema)
let create_table = Note.get("create_table")
create_table(0)

logger.info("Database initialized")

// -- Pre-compile templates -------------------------------
let index_tmpl = tmpl.from_file("templates/index.html")
let note_tmpl = tmpl.from_file("templates/note.html")

// -- Helper: parse form body -----------------------------
fn parse_form(body: String) -> Map {
    let result = Map::new()
    let pairs = body.split("&")
    for pair in pairs {
        let eq = pair.index_of("=")
        if eq > 0 {
            let key = pair.substring(0, eq)
            let val = pair.substring(eq + 1, len(pair))
            // url_decode handles both + and %XX
            result.set(url_decode(key), url_decode(val))
        }
    }
    return result
```

38.5.4 Testing It Out

Start the server:

Listing 38.46: Starting the application

```
$ clat --regvm app.lat
2026-03-14 10:04:12 [INFO] Database initialized
2026-03-14 10:04:12 [INFO] Starting Lattice Notes
Lattice HTTP server listening on http://localhost:8080
```

The two bracketed lines are our logger on stderr; the third is `app.listen` on stdout.

Open a browser to `http://localhost:8080` to see the HTML interface. Create a few notes using the form, then try the JSON API:

Listing 38.47: Testing the JSON API with curl

```
# Nothing stored yet
$ curl http://localhost:8080/api/notes
[]

# Create a note
$ curl -X POST http://localhost:8080/api/notes \
  -H "Content-Type: application/json" \
  -d '{"title":"From the API","body":"Created via curl"}'
{"id":1,"message":"Note created"}

# List all notes
$ curl http://localhost:8080/api/notes
[{"id":1,"body":"Created via curl","title":"From the API",
  "created_at":"2026-03-14 10:04:31"}]

# Get a single note
$ curl "http://localhost:8080/api/notes/one?id=1"
{"id":1,"body":"Created via curl","title":"From the API",
 "created_at":"2026-03-14 10:04:31"}

# The two error paths
$ curl "http://localhost:8080/api/notes/one"
{"error":"Missing 'id' parameter"}

$ curl "http://localhost:8080/api/notes/one?id=99"
{"error":"Note not found"}
```

Those JSON objects are wrapped here to fit the page—the server sends them on one line—and once again the key order is the Map's, not the schema's.

The HTML form path works the same way, percent-encoding and all:

Listing 38.48: Posting the form

```
$ curl -X POST http://localhost:8080/notes/create \
  -d "title=Milk+%26+Bread&body=100%25+done"

$ curl http://localhost:8080/api/notes
[{"id":1,"body":"100% done","title":"Milk & Bread", ... }]
```

`url_decode` turned `Milk+%26+Bread` back into `Milk & Bread` and `100%25+done` into `100% done`. Then, rendering the index page, the template re-escaped the ampersand:

Listing 38.49: What the browser receives

```
<h3><a href="/notes/view?id=1">Milk &amp; Bread</a></h3>
<p>100% done</p>
```

Decode on the way in, escape on the way out—the template engine handles the second half for you, as long as you use `{{ }}` and not `{{{ }}}`.

38.5.5 Design Walkthrough

Let us highlight some of the patterns at work:

1. **Templates are compiled once.** We call `tmpl.from_file` at startup and reuse the compiled templates on every request. This avoids re-parsing the template strings on each page load.
2. **Dual interface.** The same data model serves both HTML pages and a JSON API. The HTML routes render templates; the API routes call `http.json`. This dual-interface pattern is common in real applications.
3. **Middleware stacking.** The `http.logger()` middleware logs every request, `http.body_parser()` handles JSON parsing, and `http.cors()` adds CORS headers. Adding authentication would be one more `app.use` call.
4. **The ORM hides SQL.** The route handlers never write SQL directly. They interact with the `Note` model through its `create`, `find`, `all`, and `where` closures. The `where` closure's parameterized queries prevent SQL injection.
5. **Error responses are intentional.** Every API endpoint checks for missing parameters and returns structured JSON error objects with appropriate HTTP status codes. This makes the API client-friendly.
6. **Decoding happens once, at the edge.** `parse_form` is the only place that knows about percent-encoding. By the time a title reaches the ORM it is an ordinary Lattice string, and by the time it reaches the browser the template has escaped it. Every layer in between gets to be ignorant, which is the point.
7. **The model is captured, not passed.** Each route closure captures `Note`, `index_tmpl`, and `logger` from the enclosing scope rather than receiving them as parameters. The handlers only *read* those captures—`Note.get("find")` pulls a closure out and calls it—so the copy-versus-share question never arises.

One Request at a Time

`app.listen` is a single `loop` that accepts a connection, reads it, dispatches, writes, and closes—then accepts the next one. A handler that blocks for two seconds blocks the whole server for two seconds. Nothing here is thread-safe by construction, because nothing here is threaded. That is fine for a tool on your own machine and for the applications in this chapter. It is not fine for anything facing the open internet. Put a real reverse proxy in front of it, or restructure around `spawn` and a `scope` if you want concurrency—and read the concurrency chapters first, because a spawned task runs on a deep clone of what it captured.

Streaming and SSE

The `http_server` library also supports chunked transfer encoding via `http.stream(status, chunks)` and Server-Sent Events via `http.sse_stream(events)`. These are useful for long-running responses, real-time updates, and progressive data delivery. Explore `lib/http_server.lat` for the full API.

Exercises

1. **Add a DELETE endpoint.** Add `DELETE /api/notes?id=N` and `POST /notes/delete?id=N` routes to the Lattice Notes application. The HTML route should redirect back to `/` after deletion.

2. **Note editing.** Add a `GET /notes/edit?id=N` route that renders an edit form (pre-filled with the existing note data) and a `POST /notes/update` route that updates the note and redirects to the view page.

3. **Search.** Add a `GET /search?q=term` route that uses the ORM's `where` closure to find notes whose title or body contains the search term. Render the results using the index template.

4. **Authentication middleware.** Write a middleware that checks for a session cookie. If the cookie is missing, redirect HTML requests to a `/login` page and return 401 for API requests. Use `http.set_cookie` to issue the session cookie after login.

5. **Rate limiting.** Build a middleware that tracks the number of requests per client IP address (stored in a Map) and returns `429 Too Many Requests` if a client exceeds 60 requests per minute. (Hint: store timestamps and use `time()` to calculate the window, and remember that `req.get("peer")` carries the address and port together.)

6. **Case-insensitive headers.** Write a middleware that runs first, reads `req.get("headers")`, and rewrites it so every key is lowercase. Then change `require_api_key` to look for `"x-api-key"` and confirm that all three spellings of the header now work.

7. **Give them a 405.** Register your routes through a wrapper that also records which methods each path supports. Add a final catch-all with `app.all` that answers 405 with an `Allow` header when the path is known and 404 when it is not.
8. **Round-trip a note.** Write a test that creates a note through the JSON API, fetches it back, and compares the two Maps. Compare key by key rather than string-comparing the JSON: `json_stringify` emits keys in hash order, so two equal Maps can serialize to different strings.

What's Next

Congratulations—you have built both a CLI tool and a web service in Lattice, using real libraries that handle real-world concerns. You have seen how the language's Map-based closure patterns give libraries like `http_server`, `template`, and `orm` a flexible, composable feel without requiring complex type hierarchies.

These two chapters are the capstone of *The Lattice Handbook*. Everything we have covered—from the phase system and structured concurrency to modules, testing, and native extensions—comes together when you build real applications. The appendices that follow serve as a reference for the language grammar, opcode tables, and built-in functions. Keep them close while you build your next project. Happy forging.

Appendix A

Language Grammar Reference

This appendix gives a complete EBNF-style grammar for the Lattice language as of **v0.6.4**. Productions are derived directly from the recursive-descent parser in `src/parser.c` and the lexer in `src/lexer.c`. Where the grammar and the running binary could disagree, the binary won: every production below was exercised against `clat` on all three backends.

A.1 Notation

`|`	Alternation
`[ ... ]`	Optional (zero or one)
`{ ... }`	Repetition (zero or more)
`( ... )`	Grouping
`'...'`	Terminal / keyword
italic	Non-terminal

Reserved versus contextual keywords

Lattice has two tiers of keyword. *Reserved* words are recognized by `keyword_lookup()` in `src/lexer.c` and can never be used as identifiers. *Contextual* words are ordinary identifiers that the parser recognizes by spelling in one specific position — `select`, `where`, `except`, `require`, `ensure`, `timeout`, `default`, `fluid`, `crystal`, `sublimated`, and `Fn`. All of those remain perfectly good variable names. `flux select = 1` compiles and prints `1` on every backend.

A.2 Program Structure

```
Program       = [ ModeDirective ] { Item } ;
ModeDirective = '#mode' ( 'casual' | 'strict' ) ;
Item          = [ 'export' ] [ ItemAnnotation ]
                ( FnDecl | StructDecl | EnumDecl | TraitDecl
                | ImplBlock | TestDecl | Statement ) ;
ItemAnnotation = '@' ( 'fluid' | 'crystal' ) ;
```

The `@fluid` and `@crystal` annotations attach a default phase intent to a top-level item. Any other name after `@` is a parse error: `unknown annotation '@...' (expected @fluid or @crystal)`.

A.3 Declarations

```
FnDecl     = 'fn' IDENT [ TypeParams ]
             '(' [ ParamList ] ')' [ '->' TypeExpr ]
             { ContractClause }
             '{' Block '}' ;
TypeParams = '<' IDENT { ',' IDENT } '>' ;
ParamList  = Param { ',' Param } ;
Param      = [ '...' ] IDENT ':' TypeExpr [ '=' Expr ] ;

ContractClause = ( 'require' | 'ensure' ) Expr [ ',' STRING_LIT ] ;

StructDecl = 'struct' IDENT [ TypeParams ]
             '{' [ FieldDecl { ',' FieldDecl } [ ',' ] ] '}' ;
FieldDecl  = IDENT ':' TypeExpr ;

EnumDecl   = 'enum' IDENT [ TypeParams ]
             '{' [ Variant { [ ',' ] Variant } [ ',' ] ] '}' ;
Variant    = IDENT [ '(' [ TypeExpr { ',' TypeExpr } ] ')' ] ;

TraitDecl  = 'trait' IDENT '{' { TraitMethod } '}' ;
TraitMethod = 'fn' IDENT '(' [ ParamList ] ')'
              [ '->' TypeExpr ] [ ';' ] ;

ImplBlock  = 'impl' IDENT 'for' IDENT '{' { FnDecl } '}' ;

TestDecl   = 'test' STRING_LIT '{' Block '}' ;
```

A.3.1 Function contracts

A function declaration may carry any number of `require` and `ensure` clauses between the return type and the body. A `require` clause is a boolean expression checked on entry; an `ensure` clause is a *closure* that receives the return value and is checked on exit. The optional string after the comma becomes the error message.

```
fn half(n: Int) -> Int
    require n % 2 == 0, "n must be even"
    ensure |r| { r >= 0 }, "result must be non-negative"
{
    n / 2
}

print(half(10))                                  // 5
try { print(half(3)) } catch e { print(e.message) }
// require failed in 'half': n must be even
```

Impl blocks always name a trait

The `for` clause in an `impl` block is *not* optional. Writing `impl Point { ... }` fails with `parse error: expected 'for' after trait name in impl block`. Every `impl` is a trait implementation: `impl Describe for Point`.

Trait methods are signatures only

A trait method has no body. Adding one — `fn shout(self: any) -> String { "!" }` inside a `trait` — produces `parse error: expected 'fn', got '{'`. There are no default method implementations in v0.6.4.

Generic syntax parses; a bare type parameter in a signature fails at call time

`TypeParams` are accepted on functions, structs, and enums, and type arguments are accepted in type expressions. They are recorded in the AST and then ignored by the compiler — there is no type substitution, no monomorphization, and no constraint checking anywhere in the pipeline. `struct Box<T> { v: Int }` and `fn id<T>(x: Int) -> Int { x }` compile and run on all three backends, and there the `T` does nothing.

The runtime type checker is a separate mechanism, and it is not so forgiving. It validates each declared parameter type and return type at call time, and a type-parameter name is not a type it knows. So a *bare* type parameter used as a parameter or return annotation turns a well-formed function into one that raises on first call, on all three backends. Only the prefix varies (`vm error:`, `error:`, `regvm error:`), plus one detail: the register VM leaves the function name out of the return-type message.

```
fn pick<T>(x: T) -> T { return x }
print(pick(42))
// vm error: function 'pick' parameter 'x' expects type T, got Int
//           (did you mean 'Fn'?)
```

The return position fails the same way. Run this one on its own, since the program above never reaches a second statement:

```
fn wrap<T>(x: any) -> T { return x }
print(wrap(42))
// vm error: function 'wrap' return type expects T, got Int
//           (did you mean 'Fn'?)
```

Struct and enum fields are not run through that checker, so `struct Box<T> { value: T }` constructs and reads back without complaint. Treat generic syntax as forward-looking decoration, keep bare type parameters out of `fn` signatures, and never write an example that depends on generics doing work. Chapter 18 works through the boundary case by case. Generics and map/set/buffer literals are the two remaining language gaps shared by the C compiler and the self-hosted compiler.

A.4 Type Expressions

```
TypeExpr      = [ PhasePrefix ] TypeBody ;
PhasePrefix   = '~' | '*' | 'flux' | 'fix'
              | '(' PhaseConstraint ')' ;
PhaseConstraint = PhaseSymbol { '|' PhaseSymbol } ;
PhaseSymbol   = '~' | '*' | 'flux' | 'fix' | 'sublimated' ;

TypeBody      = ArrayType | FnType | NamedType ;
ArrayType     = '[' TypeExpr ']' ;
FnType        = ( 'fn' | 'Fn' ) '(' [ TypeExpr { ',' TypeExpr } ] ')'
                [ '->' TypeExpr ] ;
NamedType     = IDENT [ '<' TypeExpr { ',' TypeExpr } '>' ] ;
```

A.4.1 Function type annotations

A parameter that holds a callable can be annotated with its shape. Both spellings work, and the return arrow is optional:

```
fn apply(f: Fn(Int) -> Int, x: Int) -> Int { f(x) }
fn twice(g: fn(Int) -> Int, x: Int) -> Int { g(g(x)) }

print(apply(|n| { n + 1 }, 41))   // 42
print(twice(|n| { n * 2 }, 5))    // 20
```

`Fn` is contextual: it is treated as a function type only when the very next token is `(`. A bare `Fn` with no parameter list is parsed as an ordinary named type, which is how the older "any callable" annotation still works:

```
fn call_it(f: Fn) -> Int { f(1) }
print(call_it(|n| { n + 100 }))   // 101
```

The lowercase `fn` spelling *requires* the parenthesized parameter list — `fn` alone as a type is a parse error.

A.5 Statements

```
Block     = { Statement } ;
Statement = Binding | Assignment | ForStmt | WhileStmt
          | LoopStmt | ReturnStmt | BreakStmt | ContinueStmt
          | ImportStmt | DeferStmt | DestructureStmt
          | ExprStmt ;

Binding   = ( 'let' | 'flux' | 'fix' ) IDENT
            [ ':' TypeExpr ] '=' Expr [ ';' ] ;
Assignment = Expr AssignOp Expr [ ';' ] ;
AssignOp   = '=' | '+=' | '-=' | '*=' | '/=' | '%='
           | '&=' | '|=' | '^=' | '<<=' | '>>=' ;

ForStmt    = 'for' IDENT 'in' Expr '{' Block '}' ;
WhileStmt  = 'while' Expr '{' Block '}' ;
LoopStmt   = 'loop' '{' Block '}' ;
ReturnStmt = 'return' [ Expr ] [ ';' ] ;
BreakStmt  = 'break' [ ';' ] ;
ContinueStmt = 'continue' [ ';' ] ;

DeferStmt  = 'defer' '{' Block '}' ;

DestructureStmt = ( 'let' | 'flux' | 'fix' )
                  DestructureTarget '=' Expr [ ';' ] ;
DestructureTarget = '[' IDENT { ',' IDENT }
                    [ ',' '...' IDENT ] ']'
                  | '{' IDENT { ',' IDENT } '}' ;

ImportStmt = 'import' STRING_LIT [ 'as' IDENT ]
           | 'import' '{' IDENT { ',' IDENT } '}'
             'from' STRING_LIT ;

ExprStmt   = Expr [ ';' ] ;
```

A.5.1 Import forms

Both import forms begin with the `import` keyword. The selective form puts the brace list *before* `from`:

```
import "text_utils.lat" as text          // whole module, aliased
import { shout, quiet } from "text_utils.lat"   // selected exports
```

There is no `from . . . import . . .` statement

Starting a statement with `from` is a parse error, and because the parser recovers and keeps going you get two diagnostics for the one mistake:

```
error: 1:1: parse error: unexpected token 'from'
in expression
1:26: parse error: expected string literal or
'{' after 'import'
```

The `from` keyword only ever appears *after* a brace list in an import, or in a `select` arm.

A.6 Expressions

Lattice uses precedence-climbing for binary operators. The table below lists precedences from lowest (top) to highest (bottom), and mirrors the parser's function chain (`parse_nil_coalesce` → `parse_or` → … → `parse_postfix`).

Prec.	Category	Operators	Assoc.
1	Nil coalesce	`??`	Left
2	Logical OR	`||`	Left
3	Logical AND	`&&`	Left
4	Bitwise OR	`|`	Left
5	Bitwise XOR	`^`	Left
6	Bitwise AND	`&`	Left
7	Equality	`== !=`	Left
8	Comparison	`< > <= >=`	Left
9	Shift	`« »`	Left
10	Range	`..`	None
11	Additive	`+ -`	Left
12	Multiplicative	`* / %`	Left
13	Unary	`- ! ~`	Right
14	Postfix	`. ?. [] ?[] () ?`	Left

Table A.1: Operator precedence (lowest to highest).

```
Expr         = NilCoalesce ;
NilCoalesce  = OrExpr { '??' OrExpr } ;
OrExpr       = AndExpr { '||' AndExpr } ;
AndExpr      = BitOrExpr { '&&' BitOrExpr } ;
BitOrExpr    = BitXorExpr { '|' BitXorExpr } ;
BitXorExpr   = BitAndExpr { '^' BitAndExpr } ;
BitAndExpr   = EqualityExpr { '&' EqualityExpr } ;
EqualityExpr = CompareExpr { ( '==' | '!=' ) CompareExpr } ;
CompareExpr  = ShiftExpr { ( '<' | '>' | '<=' | '>=' ) ShiftExpr } ;
ShiftExpr    = RangeExpr { ( '<<' | '>>' ) RangeExpr } ;
RangeExpr    = AddExpr [ '..' AddExpr ] ;
AddExpr      = MulExpr { ( '+' | '-' ) MulExpr } ;
MulExpr      = UnaryExpr { ( '*' | '/' | '%' ) UnaryExpr } ;
UnaryExpr    = ( '-' | '!' | '~' ) UnaryExpr | PostfixExpr ;
```

There are no ++ or -- operators

Lattice has no increment or decrement operators. `x++` is a parse error:

```
error: 2:3: parse error: unexpected token '+' in expression
```

`x--` is worse, because it *parses* — as binary minus followed by unary minus applied to whatever comes next — and then fails at runtime with something unrelated, such as `operand must be a number for unary '-'`. Use `x += 1` and `x -= 1`. (The compilers do emit dedicated increment and decrement instructions for those forms; see Chapter B. Those are internal opcodes, not surface syntax.)

A.6.1 Postfix Expressions

```
PostfixExpr = PrimaryExpr { PostfixOp } ;
PostfixOp   = '.' IDENT [ '(' [ ArgList ] ')' ]
            | '?.' IDENT [ '(' [ ArgList ] ')' ]
            | '[' Expr ']'
            | '?[' Expr ']'
            | '(' [ ArgList ] ')'     (* call *)
            | '?'                     (* try/propagate *)
            ;
ArgList     = Expr { ',' Expr } ;
```

A.6.2 Primary Expressions

```
PrimaryExpr = INT_LIT | FLOAT_LIT | STRING_LIT
            | InterpString
            | 'true' | 'false' | 'nil'
            | IDENT
            | StructLiteral
            | EnumVariant
            | ArrayLiteral
            | TupleLiteral
            | GroupedExpr
            | BlockExpr
            | ClosureExpr
            | IfExpr
            | MatchExpr
            | TryCatchExpr
            | SelectExpr
            | PhaseExpr
            | PrintExpr
            | SpawnExpr
            | ScopeExpr ;

InterpString  = INTERP_START Expr
                { INTERP_MID Expr } INTERP_END ;
StructLiteral = IDENT '{' [ FieldInit { ',' FieldInit } ] '}' ;
FieldInit     = IDENT ':' Expr ;
EnumVariant   = IDENT '::' IDENT
                [ '(' [ ArgList ] ')' ] ;
ArrayLiteral  = '[' [ ArrayElem { ',' ArrayElem } ] ']' ;
ArrayElem     = [ '...' ] Expr ;
TupleLiteral  = '(' Expr ',' [ Expr { ',' Expr } ] ')' ;
GroupedExpr   = '(' Expr ')' ;
BlockExpr     = '{' Block '}' ;
ClosureExpr   = '|' [ ClosureParam { ',' ClosureParam } ] '|'
                Expr ;
ClosureParam  = [ '...' ] IDENT [ '=' Expr ] ;
```

Closure parameters take no type annotations

This is the mirror image of function declarations, and it trips up nearly everyone once the `Fn(...) -> T` annotations exist.

	`fn` parameters	Closure parameters
Type annotation	**required**	**forbidden**
Default value	allowed	allowed
Variadic `...`	allowed (last only)	allowed (last only)

Omitting the annotation on an `fn` parameter fails with `parse error: expected ':', got ')'`. Adding one to a closure parameter fails with `parse error: expected ',', got ':'`. Two more closure details. An empty parameter list is written `| |` with a space — `||` lexes as logical OR and yields `parse error: unexpected token '||' in expression`. And a default-value expression is parsed at bitwise-XOR precedence, so a bare `|` inside one would be read as the closing delimiter; parenthesize if you need it.

A.6.3 Control-Flow Expressions

```
IfExpr       = 'if' Expr '{' Block '}'
               [ 'else' ( IfExpr | '{' Block '}' ) ] ;

MatchExpr    = 'match' Expr '{'
               { [ PhaseQual ] Pattern
                 [ 'if' Expr ] '=>'
                 ( '{' Block '}' | Expr ) [','] }
               '}' ;
PhaseQual    = 'fluid' | 'crystal' ;

TryCatchExpr = 'try' '{' Block '}'
               'catch' IDENT '{' Block '}' ;

SelectExpr   = 'select' '{' { SelectArm [','] } '}' ;
SelectArm    = IDENT 'from' Expr '=>' '{' Block '}'
             | 'timeout' '(' Expr ')' '=>' '{' Block '}'
             | 'default' '=>' '{' Block '}' ;
```

`select` is contextual: the parser only treats it as a select expression when the very next token is `{`. Select arm bodies must be braced blocks; unlike match arms, a bare expression body is not accepted.

A.6.4 Phase Expressions

```
PhaseExpr     = FreezeExpr | ThawExpr | CloneExpr
              | AnnealExpr | ForgeExpr | CrystallizeExpr
              | BorrowExpr | SublimateExpr ;

FreezeExpr    = 'freeze' '(' Expr ')'
                [ 'where' Expr ]
                [ 'except' '[' Expr { ',' Expr } ']' ] ;
ThawExpr      = 'thaw' '(' Expr ')' ;
CloneExpr     = 'clone' '(' Expr ')' ;
AnnealExpr    = 'anneal' '(' Expr ')' ClosureExpr ;
ForgeExpr     = 'forge' '{' Block '}' ;
CrystallizeExpr = 'crystallize' '(' Expr ')' '{' Block '}' ;
BorrowExpr    = 'borrow' '(' Expr ')' '{' Block '}' ;
SublimateExpr = 'sublimate' '(' Expr ')' ;
```

A.6.5 The freeze modifiers

`freeze` accepts two optional trailing clauses, in this order: `where` first, then `except`. Both are contextual identifiers, so `where` and `except` remain usable as variable names elsewhere.

The `where` clause attaches a contract closure that must accept the value being crystallized:

```
flux temps = [1, 2, 3]
freeze(temps) where |v| { v.len() == 3 }
print(phase_of(temps))            // crystal
```

The `except` clause takes a bracketed list of key expressions and punches fluid holes in an otherwise-crystal aggregate. Writes to an exempt key succeed; writes to any other key raise `cannot modify a frozen value`.

```
flux config = Map::new()
config["host"] = "example.com"
config["retries"] = 1

freeze(config) except ["retries"]

config["retries"] = 5
print(config["retries"])          // 5
config.set("retries", 7)
print(config["retries"])          // 7
config.remove("retries")
print(config.has("retries"))      // false

try { config["host"] = "y" } catch e { print(e.message) }
// cannot modify a frozen value
```

Method mutation of exempt keys

As of v0.6.4 the exemption covers method mutators too — `set` and `remove` on an exempt key both succeed, where earlier releases only allowed index assignment. Whole-container mutators still raise, because they do not target an exempt key — `config.merge(other)` gives `cannot call mutating method 'merge' on crystal value 'config' (use thaw(config) to make it mutable)`. The behaviour above is identical on the stack VM, `--tree-walk`, and `--regvm`. See Chapter 11 for the full semantics, including the rule that a plain `freeze()` afterwards closes every hole again.

A.6.6 Other Expressions

```
PrintExpr = 'print' '(' [ ArgList ] ')' ;
SpawnExpr = 'spawn' '{' Block '}' ;
ScopeExpr = 'scope' '{' Block '}' ;
```

A.7 Patterns

Patterns appear in `match` arms and destructuring bindings.

```
Pattern    = LitPat | WildcardPat | BindingPat
           | RangePat | ArrayPat | StructPat
           | EnumPat ;
LitPat     = [ '-' ] ( INT_LIT | FLOAT_LIT )
           | STRING_LIT | 'true' | 'false' | 'nil' ;
WildcardPat = '_' ;
BindingPat  = IDENT ;
RangePat    = ( INT_LIT | IDENT ) '..' Expr ;
ArrayPat    = '[' [ ArrayPatElem { ',' ArrayPatElem } [ ',' ] ] ']' ;
ArrayPatElem = [ '...' ] Pattern ;
StructPat   = '{' [ StructPatField { ',' StructPatField } [ ',' ] ] '}' ;
StructPatField = IDENT [ ':' Pattern ] ;
EnumPat     = IDENT '::' IDENT
              [ '(' [ Pattern { ',' Pattern } ] ')' ] ;
```

The `-` prefix in `LitPat` applies only to numeric literals; `expected number after '-' in pattern` is the error otherwise. There are no or-patterns (`A | B`) and no binding-with-subpattern (`name @ pat`).

A.7.1 Enum variant patterns

A `match` arm can name an enum variant and destructure its payload in place. The payload slots are full patterns, so bindings, wildcards, and literals all nest freely, and a unit variant omits the parentheses entirely.

```
enum Shape { Circle(Float), Rect(Float, Float), Point }

fn area(s: any) -> Float {
    match s {
        Shape::Circle(r)   => 3.14 * r * r,   // payload binding
        Shape::Rect(_, h)  => h * 2.0,        // wildcard sub-pattern
        Shape::Point       => 0.0,            // unit variant
        _                  => -1.0,
    }
}

print(area(Shape::Circle(1.0)))   // 3.14
print(area(Shape::Rect(3.0, 4.0)))// 8
print(area(Shape::Point))         // 0
```

Literal sub-patterns let us branch on the payload value itself, and arms are tried top to bottom, so put the specific ones first:

```
enum Code { Status(Int), Msg(String) }

fn describe(c: any) -> String {
    match c {
        Code::Status(200) => "ok",
        Code::Status(404) => "missing",
        Code::Status(n)   => "status ${n}",
        Code::Msg("hi")   => "greeting",
        Code::Msg(m)      => "msg ${m}",
        _                 => "other",
    }
}

print(describe(Code::Status(200)))  // ok
print(describe(Code::Status(500)))  // status 500
print(describe(Code::Msg("bye")))   // msg bye
```

All of the above behaves identically on the stack VM, `--tree-walk`, and `--regvm`.

A.8 Lexical Grammar

A.8.1 Keywords

All reserved keywords recognized by `keyword_lookup()` in `src/lexer.c` — forty of them:

```
let  flux  fix  fn  struct  enum  trait  impl  if  else  for  in  while
loop  return  break  continue  match  true  false  nil  freeze  thaw
clone  anneal  forge  crystallize  borrow  sublimate  spawn  scope  try
catch  defer  import  from  as  export  print  test
```

The contextual words listed in the notation section — notably `select`, `where`, `except`, `require`, `ensure`, `timeout`, `default`, `fluid`, `crystal`, `sublimated`, and `Fn` — are deliberately *not* on this list. They lex as ordinary identifiers.

A.8.2 Literals

```
INT_LIT    = DIGIT { DIGIT | '_' }
           | ( '0x' | '0X' ) HEXDIGIT { HEXDIGIT | '_' } ;
FLOAT_LIT  = DIGIT { DIGIT | '_' } '.' DIGIT { DIGIT | '_' } ;
STRING_LIT = '"' { CHAR | EscapeSeq | '${' Expr '}' } '"'
           | '"""' { CHAR | EscapeSeq | '${' Expr '}' } '"""' ;
EscapeSeq  = '\' ( 'n' | 't' | 'r' | '0' | '\' | '"' | ''' | '$'
             | 'x' HEXDIGIT HEXDIGIT
             | ANY ) ;
IDENT      = ( ALPHA | '_' ) { ALPHA | DIGIT | '_' } ;
```

Numeric literals: what is *not* there

The lexer supports exactly two integer bases and one float shape. In particular:

- There are **no binary or octal literals**. `0b1010` and `0o17` both lex as the integer `0` followed by an identifier, producing `parse error: expected ',', got 'identifier'`. A leading zero is ignored, so `017` is decimal 17, not octal.
- There is **no scientific notation**. `1.5e3` fails with `parse error: expected ',', got 'e3'`. Write `1.5 * 1000.0` or a plain decimal.
- A float needs digits on both sides of the dot: `.5` is a parse error.
- Underscores are allowed *anywhere* among the digits and are stripped before conversion: `1_000_000` is `1000000`, and `1_0.2_5` is `10.25`.
- An integer literal that will not fit in a signed 64-bit value is a compile-time error: `error: 1:7: integer literal out of range`. There is no silent saturation.

Escape sequences and the missing NUL

The recognized escapes are `\n`, `\t`, `\r`, `\0`, `\\`, `\"`, `\'`, `\$`, and `\xHH`. Anything else after a backslash yields the character itself with the backslash dropped, so `"\e"` yields `"e"` — unknown escapes are silent, not errors.

The `\0` case deserves care. The lexer does produce a NUL byte, but the literal is carried as a C string from there on, so the NUL terminates it. `"\0".len()` is `0`, and `"A\0B".len()` is `1`. `"\x00"` behaves the same way. Lattice strings themselves *are* binary-safe in v0.6.4 — embedded NULs survive equality, slicing, splitting, and `len()` — but they have to arrive through a `Buffer`, `json_parse`, a socket, or process output rather than through a string literal.

Triple-quoted strings (`"""..."""`) are dedented automatically against the least-indented content line, and support the same escapes and `${...}` interpolation as ordinary strings.

A.8.3 Operators and Delimiters

Token	Symbol	Token	Symbol	Token	Symbol
`+`	Plus	`==`	Equal	`(`	LParen
`-`	Minus	`!=`	NotEqual	`)`	RParen
`*`	Star	`<`	Less	`{`	LBrace
`/`	Slash	`>`	Greater	`}`	RBrace
`%`	Percent	`<=`	LessEq	`[`	LBracket
`=`	Assign	`>=`	GreaterEq	`]`	RBracket
`!`	Bang	`&&`	LogicAnd	`,`	Comma
`~`	Tilde	`||`	LogicOr	`:`	Colon
`.`	Dot	`&`	Ampersand	`::`	ColonColon
`..`	DotDot	`|`	Pipe	`;`	Semicolon
`...`	DotDotDot	`^`	Caret	`@`	At
`->`	Arrow	`«`	LShift	`??`	NilCoalesce
`=>`	FatArrow	`»`	RShift	`?.`	OptChainDot
`?`	Question	`?[`	OptChainIdx		

Table A.2: Operators and delimiters.

A.8.4 Compound Assignment Operators

All compound assignment operators desugar in the parser to the corresponding binary operation followed by assignment — `target op= rhs` becomes `target = target op rhs`:

```
+=   -=   *=   /=   %=   &=   |=   ^=   «=   »=
```

Because the desugaring duplicates the target expression, a target that cannot be cloned is rejected at parse time with `invalid compound assignment target`.

A.8.5 Comments

```
LineComment  = '//' { ANY } NEWLINE ;
DocComment   = '///' { ANY } NEWLINE ;
BlockComment = '/*' { ANY | BlockComment } '*/' ;
```

Block comments **nest**: the lexer tracks depth, so `/* outer /* inner */ still comment */` is one comment and the code after it compiles normally. Triple-slash `///` comments are ordinary line comments to the lexer, but the formatter and `clat doc` treat them as documentation and preserve them.

Comments and whitespace (spaces, tabs, newlines, carriage returns) are discarded by the lexer and do not appear in the grammar productions above. Semicolons are optional statement terminators; the parser consumes them when present but does not require them.

A.9 Parser Limits

The parser guards against pathological input with a shared depth counter (`PARSER_MAX_DEPTH`) covering expressions, patterns, and type expressions, plus a separate interpolation limit. Exceeding them is a clean compile error rather than a crash:

Limit	Message
Expression nesting, 1000	`expression nesting too deep`
Pattern nesting, 1000	`pattern nesting too deep`
Interpolation nesting, 200	`string interpolation nested too deep`
Errors reported per run, 10	— (parsing stops after ten)

The parser does *not* stop at the first error. Panic-mode recovery resynchronizes at statement and item boundaries and collects up to ten diagnostics per run, which is why the samples in this appendix sometimes show two error lines for a single mistake.

Appendix B

Opcode Reference

This appendix is the authoritative listing of every bytecode instruction for both the Stack VM and Register VM backends, reconciled against `include/stackopcode.h`, `include/regopcode.h`, the two compilers, and the two verifiers as of **v0.6.4**. Opcodes are grouped by category and listed in their enum order.

Opcode numbers are part of the file format

The numeric value of an opcode is its position in the enum, and serialized `.latc` and `.rlat` files store raw opcode bytes. That makes the numbering a compatibility surface: new instructions are *appended* to the end of the enum rather than slotted in beside their relatives, so already-built bytecode — and the self-hosted compiler in `compiler/latc.lat`, which hardcodes opcode numbers — keeps working. This is why the tail of each enum reads like an archaeological dig. The tables below are grouped by *topic* for readability; Table B.1 gives the numeric anchors.

There is no bytecode-dump flag

`clat --help` lists no disassembly option, and nothing in the shipped CLI calls the disassembler. The routines are there — `chunk_disassemble()` and `chunk_disassemble_instruction()` in `src/chunk.c`, declared in `include/chunk.h` — and they print to stderr, so an embedding host can call them directly. To inspect bytecode from the command line today, the practical route is `clat compile prog.lat -o prog.latc` followed by a hex dump of the result, using Table B.1 to orient yourself.

#	Stack VM	#	Register VM
0	OP_CONSTANT	0	ROP_MOVE
83	OP_SCOPE	80	ROP_RESET_EPHEMERAL
84	OP_SELECT	100	ROP_HALT
93	OP_CONSTANT_16	101	ROP_GETFIELD_16
100	OP_RESET_EPHEMERAL	103	ROP_INVOKE_MUT
113	OP_HALT	105	ROP_SELECT_CHOOSE
114	OP_INDEX_GLOBAL	110	ROP_INVOKE_UPVALUE
116	OP_INVOKE_MUT	111	ROP_COUNT (total)
124	OP_INVOKE_UPVALUE_16		

Table B.1: Numeric landmarks. The Stack VM defines 125 opcodes (0–124); the Register VM defines 111 (0–110).

B.1 Stack VM Opcodes

The Stack VM uses variable-length instructions: a one-byte opcode followed by zero or more operand bytes (typically one- or two-byte unsigned indices, big-endian where two bytes are used). All arithmetic pops its operands from the stack and pushes the result. The **Size** column is the total instruction length in bytes, opcode included — it is exactly what `verify_instr_length()` in `src/chunk.c` enforces when loading untrusted bytecode.

B.1.1 Stack Manipulation

Opcode	Operands	Size	Description
OP_CONSTANT	idx (1B)	2	Push `constants[idx]` onto the stack.
OP_CONSTANT_16	idx (2B)	3	Wide variant—16-bit big-endian constant index.
OP_NIL	—	1	Push `nil`.
OP_TRUE	—	1	Push `true`.
OP_FALSE	—	1	Push `false`.
OP_UNIT	—	1	Push `unit`.
OP_POP	—	1	Discard top of stack.
OP_DUP	—	1	Duplicate top of stack.
OP_SWAP	—	1	Swap the top two stack values.
OP_LOAD_INT8	int8 (1B)	2	Push small integer literal from the instruction stream.

Opcode	Operands	Size	Description
OP_ADD	—	1	Pop two values, push their sum (+). Also concatenates strings.
OP_SUB	—	1	Subtraction (-).
OP_MUL	—	1	Multiplication (*).
OP_DIV	—	1	Division (/).
OP_MOD	—	1	Modulo (%).
OP_NEG	—	1	Negate top of stack.
OP_NOT	—	1	Logical NOT (!).
OP_CONCAT	—	1	String concatenation (interpolation lowering).

B.1.2 Arithmetic and Logic

Where the ephemeral arena comes in

Both of these opcodes join strings, but they allocate differently. OP_CONCAT — the lowering of string interpolation — builds its result straight into the VM's ephemeral bump arena (stackvm_ephemeral_concat over vm->ephemeral in src/stackvm.c), skipping malloc entirely. OP_ADD's string path mallocs, then interns short results for pointer-equality comparisons; it also has a realloc-in-place fast path that turns the common s += "x" loop from $O(n)$ per append into amortized $O(1)$.

The arena is wiped by OP_RESET_EPHEMERAL, which the compiler emits after every statement. Before resetting, the VM walks the whole stack and *promotes* any arena-backed value to the heap, and it promotes a frame's ephemerals again on the way into a call. See include/memory.h and src/memory.c for the arena itself.

B.1.3 Bitwise Operations

Opcode	Operands	Size	Description
OP_BIT_AND	—	1	Bitwise AND (&).
OP_BIT_OR	—	1	Bitwise OR (|).
OP_BIT_XOR	—	1	Bitwise XOR (^).
OP_BIT_NOT	—	1	Bitwise NOT (~).
OP_LSHIFT	—	1	Left shift («).
OP_RSHIFT	—	1	Right shift (»).

Opcode	Operands	Size	Description
`OP_EQ`	—	1	Pop two values, push `true` if equal.
`OP_NEQ`	—	1	Not equal.
`OP_LT`	—	1	Less than.
`OP_GT`	—	1	Greater than.
`OP_LTEQ`	—	1	Less than or equal.
`OP_GTEQ`	—	1	Greater than or equal.

Opcode	Operands	Size	Description
`OP_GET_LOCAL`	slot (1B)	2	Push local from stack slot.
`OP_SET_LOCAL`	slot (1B)	2	Set local at slot; keep value on stack.
`OP_SET_LOCAL_POP`	slot (1B)	2	Set local at slot and pop the value.
`OP_GET_GLOBAL`	idx (1B)	2	Push `globals[constants[idx]]`.
`OP_GET_GLOBAL_16`	idx (2B)	3	Wide variant.
`OP_SET_GLOBAL`	idx (1B)	2	Set global named by `constants[idx]`.
`OP_SET_GLOBAL_16`	idx (2B)	3	Wide variant.
`OP_DEFINE_GLOBAL`	idx (1B)	2	Define new global, pop value.
`OP_DEFINE_GLOBAL_16`	idx (2B)	3	Wide variant.
`OP_GET_UPVALUE`	idx (1B)	2	Push captured upvalue.
`OP_SET_UPVALUE`	idx (1B)	2	Set captured upvalue from TOS.
`OP_CLOSE_UPVALUE`	—	1	Close the upvalue for the local on top of stack.
`OP_CLOSE_UPVALUE_PRESERVE`	—	1	Close the local *below* TOS and remove it, keeping TOS in place.

B.1.4 Comparison

B.1.5 Variables and Upvalues

The five wide variants

A chunk's constant pool can hold more than 256 entries, and the compact opcodes carry a single-byte index. Five wide forms cover the overflow, each taking a two-byte big-endian index: `OP_CONSTANT_16`, `OP_GET_GLOBAL_16`, `OP_SET_GLOBAL_16`, `OP_DEFINE_GLOBAL_16`, and `OP_CLOSURE_16`. Method invocation has its own wide pair, `OP_INVOKE_LOCAL_16` and `OP_INVOKE_GLOBAL_16`, plus `OP_INVOKE_UPVALUE_16`. The compiler emits the compact form whenever the index fits, so wide opcodes only show up in large modules.

Opcode	Operands	Size	Description
`OP_JUMP`	off (2B)	3	Unconditional forward jump.
`OP_JUMP_IF_FALSE`	off (2B)	3	Jump if TOS is falsy (does not pop).
`OP_JUMP_IF_TRUE`	off (2B)	3	Jump if TOS is truthy (does not pop).
`OP_JUMP_IF_NOT_NIL`	off (2B)	3	Jump if TOS is not nil (does not pop).
`OP_LOOP`	off (2B)	3	Unconditional backward jump.

Opcode	Operands	Size	Description
`OP_CALL`	argc (1B)	2	Call function on stack with `argc` arguments.
`OP_CLOSURE`	idx (1B), uvc (1B), then 2×uvc	$3 + 2u$	Create closure from function constant; each upvalue contributes an `(is_local, index)` byte pair.
`OP_CLOSURE_16`	idx (2B), uvc (1B), then 2×uvc	$4 + 2u$	Wide variant—16-bit function constant index.
`OP_RETURN`	—	1	Return TOS (or unit).

B.1.6 Control Flow

B.1.7 Functions and Closures

B.1.8 Iterators

Opcode	Operands	Size	Description
`OP_ITER_INIT`	—	1	Convert range/array to iterator state.
`OP_ITER_NEXT`	off (2B)	3	Push next value or jump if exhausted.

B.1.9 Data Structures

Element-level global access

`OP_INDEX_GLOBAL` and `OP_SET_INDEX_GLOBAL` exist to kill an $O(n^2)$: without them, touching one element of a global container meant loading — and therefore deep-cloning — the entire container. These two borrow the value in the environment and address a single element. Their operand is a *two-byte* name-constant index even though they sit at opcode 114 and 115, past `OP_HALT`; they were appended so existing opcode numbers stayed put for the self-hosted compiler.

Opcode	Operands	Size	Description
OP_BUILD_ARRAY	count (1B)	2	Pop *count* values, build array.
OP_ARRAY_FLATTEN	—	1	Flatten one level of nested arrays (spread).
OP_BUILD_MAP	count (1B)	2	Pop *count*×2 values (key/val pairs), build map.
OP_BUILD_TUPLE	count (1B)	2	Pop *count* values, build tuple.
OP_BUILD_STRUCT	name, fields (2B)	3	Pop field values, build struct.
OP_BUILD_RANGE	—	1	Pop end, start; push range.
OP_BUILD_ENUM	enum, var, argc (3B)	4	Build enum with payload.
OP_INDEX	—	1	Pop index & object; push `object[index]`.
OP_INDEX_LOCAL	slot (1B)	2	Pop index; push `local[index]`.
OP_INDEX_GLOBAL	name (2B)	3	Pop index; push `global[index]`, borrowing the global in place.
OP_SET_INDEX	—	1	Pop value, index, object; set `object[index]`.
OP_SET_INDEX_LOCAL	slot (1B)	2	Pop value, index; mutate local in-place.
OP_SET_INDEX_GLOBAL	name (2B)	3	Pop index, value; mutate `global[index]` in place.
OP_GET_FIELD	idx (1B)	2	Pop object; push named field.
OP_GET_FIELD_LOCAL	slot, idx (2B)	3	Push `local[slot].field`; the local stays in its slot.
OP_SET_FIELD	idx (1B)	2	Pop value, pop object; set named field.
OP_HAS_FIELD	idx (1B)	2	Pop value; push whether a Struct/Map contains that field.
OP_SET_SLICE	—	1	Splice `obj[start..end] = val`.
OP_SET_SLICE_LOCAL	slot (1B)	2	In-place splice on local.

B.1.10 Method Invocation

`OP_INVOKE_MUT` and the write-back chain

This is the instruction that made `obj[0].push(9)` and `m["k"].push(2)` actually mutate their containers. It is emitted only when the receiver is an index expression and the method name passes a mutator prefilter. Its layout is `[method_idx:u8][arg_count:u8][skip_len:u16 BE]`, and the compiler emits a write-back chain of `skip_len` bytes immediately afterwards.

At run time the VM consults `builtin_method_mutates()`. If the call really is an in-place builtin mutation, it runs on the cloned receiver, leaves `[result, mutated_clone]` on the stack, and falls through into the write-back chain — which stores the clone back and checks the parent's phase, so a frozen parent raises rather than silently swallowing the change. If it is anything else (a non-mutating

Opcode	Operands	Size	Description
OP_INVOKE	name, argc (2B)	3	Invoke method on TOS with args.
OP_INVOKE_LOCAL	slot, name, argc (3B)	4	Invoke method on local; mutate in-place.
OP_INVOKE_LOCAL_16	slot, name16, argc (4B)	5	Wide method-name variant.
OP_INVOKE_GLOBAL	gname, mname, argc (3B)	4	Invoke method on global; write back.
OP_INVOKE_GLOBAL_16	gname16, mname16, argc (5B)	6	Wide variant.
OP_INVOKE_UPVALUE	uv, name, argc (3B)	4	Invoke method through an upvalue cell; mutations persist through the capture.
OP_INVOKE_UPVALUE_16	uv, name16, argc (4B)	5	Wide variant.
OP_INVOKE_MUT	name, argc (2B), skip (2B)	5	Nested-element mutating call; see below.

builtin, a struct or map closure field, an `impl` method, or a missing method), the VM does `frame->ip += skip_len` to step past the now-dead chain and resolves exactly like `OP_INVOKE`.

`OP_INVOKE_UPVALUE` solves the sibling problem for closures: reading a captured container with `OP_GET_UPVALUE` produces a copy, so a `push` on it used to be discarded. Invoking through the upvalue cell (`ObjUpvalue->location` — the enclosing frame's slot while open, the cell's own storage once closed) keeps the mutation where the capture can see it.

B.1.11 Exception Handling and Defer

Opcode	Operands	Size	Description
OP_PUSH_EXCEPTION_HANDLER	off (2B)	3	Push handler; catch block at `ip+off`.
OP_POP_EXCEPTION_HANDLER	—	1	Pop handler on success.
OP_THROW	—	1	Pop error value; unwind to nearest handler.
OP_TRY_UNWRAP	—	1	? operator: unwrap ok or propagate error.
OP_DEFER_PUSH	depth (1B), reserved (2B)	4	Pop the compiled cleanup closure and register it for the current frame at *depth*.
OP_DEFER_RUN	min_depth (1B)	2	Run this frame's pending defers with `scope_depth` $\geq$ `min_depth`, LIFO, preserving TOS.

The disassembler disagrees about defer

`chunk_disassemble_instruction()` still prints `OP_DEFER_PUSH` as a 3-byte jump and `OP_DEFER_RUN` as operand-free. The compiler (`src/stackcompiler.c`) and the verifier (`verify_instr_length()`) agree on 4 and 2 bytes respectively, and the VM reads a `READ_BYTE()` plus a reserved `READ_U16()` for `OP_DEFER_PUSH` and a `READ_BYTE()` for `OP_DEFER_RUN`. The sizes in the table above are the ones the machine actually uses; if you hand-decode a dump, do not trust those two disassembler lines.

B.1.12 Phase System

Opcode	Operands	Size	Description
`OP_FREEZE`	—	1	Pop value; push frozen (crystal) copy.
`OP_THAW`	—	1	Pop value; push thawed (fluid) copy.
`OP_CLONE`	—	1	Pop value; push deep clone.
`OP_MARK_FLUID`	—	1	Tag TOS FLUID; privatizes a crystal first.
`OP_SUBLIMATE`	—	1	Tag TOS SUBLIMATED.
`OP_FREEZE_VAR`	name, loc, slot (3B)	4	Freeze named variable; fire reactions.
`OP_THAW_VAR`	name, loc, slot (3B)	4	Thaw named variable; fire reactions.
`OP_SUBLIMATE_VAR`	name, loc, slot (3B)	4	Sublimate named variable.
`OP_IS_CRYSTAL`	—	1	Pop value; push bool (phase is crystal?).
`OP_IS_FLUID`	—	1	Pop value; push bool (phase is fluid?).
`OP_FREEZE_FIELD`	name, loc, slot (3B)	4	Pop key string; freeze that one field/key.
`OP_FREEZE_EXCEPT`	name, loc, slot, n (4B)	5	Pop *n* key strings; freeze everything except those.
`OP_REACT`	name (1B)	2	Pop callback; register reaction on the named variable.
`OP_UNREACT`	name (1B)	2	Remove reaction.
`OP_BOND`	target (1B)	2	Pop dependency and strategy strings; create bond.
`OP_UNBOND`	target (1B)	2	Pop dependency string; remove bond.
`OP_SEED`	name (1B)	2	Pop contract; register seed.
`OP_UNSEED`	name (1B)	2	Remove seed contract.

`OP_REACT` through `OP_UNSEED` have no dedicated syntax — the compilers pattern-match calls named `react`, `bond`, `seed` and friends whose first argument is a bare identifier, and lower them to these instructions.

Opcode	Operands	Size	Description
OP_MATCH_TYPE	ValueType (1B)	2	Keep the candidate on TOS and push an exact runtime-tag match.
OP_MATCH_RANGE	—	1	Pop end and start, preserving the candidate below them; push inclusive integer range match.
OP_MATCH_FLUID	—	1	Pop value; push whether its phase is fluid *or* unphased.

B.1.13 Pattern Matching

These three, along with OP_HAS_FIELD and OP_CLOSE_UPVALUE_PRESERVE, are what let recursive patterns — including the enum-variant payload patterns in Chapter A — lower to bytecode instead of falling back to the tree-walker. That lowering is also why `match` survives serialization in format v3.

B.1.14 Builtins, I/O, and Modules

Opcode	Operands	Size	Description
OP_PRINT	argc (1B)	2	Pop *argc* values; print them.
OP_IMPORT	idx (1B)	2	Import module at constants[idx].

B.1.15 Concurrency

Opcode	Operands	Size	Description
OP_SCOPE	spawns, sync_idx, then one idx per spawn	$3 + s$	Run s spawn sub-chunks under a structured scope, then the sync body.
OP_SELECT	arms, then 4B per arm	$2 + 4a$	Legacy select; retained for older bytecode.
OP_SELECT_CHOOSE	arms, then 1 flag byte each	$2 + a$	Choose among pre-evaluated arm operands.

Variable-length encodings

OP_SCOPE and OP_SELECT are the two classic variable-length instructions, and their operands are sub-chunk indices into the constant pool. Each referenced constant is a VAL_CLOSURE whose body is NULL and whose native_fn field holds a Chunk * — that is how a spawn body or a select arm becomes bytecode instead of an AST pointer. chunk_free recurses into them.

```
OP_SCOPE  spawn_count sync_idx spawn_idx[0] .. spawn_idx[n-1]
OP_SELECT arm_count  (flags chan_idx body_idx binding_idx) x arm_count
OP_SELECT_CHOOSE arm_count flags[0] .. flags[n-1]
```

In `OP_SELECT_CHOOSE` the flag bits are `0x01` for a `default` arm, `0x02` for a `timeout(ms)` arm, and `0x04` when the arm binds a received value. Its operands — one Channel, timeout integer, or unit per arm — are evaluated in source order before the instruction runs, and it pushes a selected-index/value pair.

`OP_SELECT` is a fossil

Today's compiler lowers every `select` to `OP_SELECT_CHOOSE`: `src/stackcompiler.c` contains exactly one select emission and it is the newer opcode. `OP_SELECT` remains implemented in `src/stackvm.c` so that bytecode produced by older releases still runs. The same is true of `ROP_SELECT` on the register side. `OP_SCOPE`, by contrast, is very much alive — it is still what `scope { spawn { ... } }` compiles to.

B.1.16 Optimization Fast-Path

Opcode	Operands	Size	Description
`OP_ADD_INT`	—	1	Integer-only add (no type check).
`OP_SUB_INT`	—	1	Integer-only subtract.
`OP_MUL_INT`	—	1	Integer-only multiply.
`OP_LT_INT`	—	1	Integer-only less-than.
`OP_LTEQ_INT`	—	1	Integer-only less-than-or-equal.
`OP_INC_LOCAL`	slot (1B)	2	Increment integer local by 1 in-place.
`OP_DEC_LOCAL`	slot (1B)	2	Decrement integer local by 1 in-place.
`OP_APPEND_STR_LOCAL`	slot (1B)	2	Pop string; append to `local[slot]` in-place via realloc.

The `_INT` opcodes skip the type check

`OP_ADD_INT` and its siblings assume both operands are already integers. The compilers only emit them where that is provable — literal integer expressions, induction variables — which is why they are safe. They are the reason a five-million-iteration counting loop runs roughly a third faster than the generic path. `OP_INC_LOCAL` and `OP_DEC_LOCAL` are the lowering of `x += 1` and `x -= 1` on an integer local; they are *not* evidence of a `++` operator, which the language does not have.

B.1.17 Type Checking, Arena, and Halt

Opcode	Operands	Size	Description
OP_CHECK_TYPE	slot, type, err (3B)	4	Throw if the local's runtime type does not match.
OP_CHECK_RETURN_TYPE	type, err (2B)	3	Throw if TOS type does not match.
OP_RESET_EPHEMERAL	—	1	Reset the ephemeral bump arena (opcode 100).
OP_HALT	—	1	Stop execution and return unit (opcode 113).

B.2 Register VM Opcodes

The Register VM uses fixed-width 32-bit instructions with four encoding formats. Instructions that need more room — the invoke family, FREEZE_EXCEPT, CHECK_TYPE, the wide field accessors — occupy two or three consecutive words, with the trailing words carrying data rather than an opcode.

Format	Layout (bits)	Use
ABC	[op:8][A:8][B:8][C:8]	Three-address operations
ABx	[op:8][A:8][Bx:16]	Constant/global access (unsigned)
AsBx	[op:8][A:8][sBx:16]	Conditional jumps (signed offset)
sBx	[op:8][sBx:24]	Unconditional jumps (signed, biased)

Table B.2: Register VM instruction encoding formats.

Notation: R[x] = register x; K[x] = constant pool entry x.

B.2.1 Data Movement

Opcode	Format	Operands	Semantics
MOVE	ABC	A, B	R[A] = R[B]
LOADK	ABx	A, Bx	R[A] = K[Bx]
LOADI	AsBx	A, sBx	R[A] = int(sBx)
LOADNIL	A	A	R[A] = nil
LOADTRUE	A	A	R[A] = true
LOADFALSE	A	A	R[A] = false
LOADUNIT	A	A	R[A] = unit

Opcode	Fmt	Operands	Semantics
ADD	ABC	A, B, C	R[A] = R[B] + R[C]
SUB	ABC	A, B, C	R[A] = R[B] - R[C]
MUL	ABC	A, B, C	R[A] = R[B] * R[C]
DIV	ABC	A, B, C	R[A] = R[B] / R[C]
MOD	ABC	A, B, C	R[A] = R[B] % R[C]
NEG	ABC	A, B	R[A] = -R[B]
ADDI	ABC	A, B, C	R[A] = R[B] + signext(C)
CONCAT	ABC	A, B, C	R[A] = str(R[B]) ++ str(R[C])

Opcode	Fmt	Operands	Semantics
EQ	ABC	A, B, C	R[A] = (R[B] == R[C])
NEQ	ABC	A, B, C	R[A] = (R[B] != R[C])
LT	ABC	A, B, C	R[A] = (R[B] < R[C])
LTEQ	ABC	A, B, C	R[A] = (R[B] <= R[C])
GT	ABC	A, B, C	R[A] = (R[B] > R[C])
GTEQ	ABC	A, B, C	R[A] = (R[B] >= R[C])
NOT	ABC	A, B	R[A] = !R[B]

B.2.2 Arithmetic

B.2.3 Comparison and Logic

B.2.4 Bitwise Operations

B.2.5 Control Flow

B.2.6 Variables, Fields, and Upvalues

B.2.7 Functions

B.2.8 Data Structures

B.2.9 Builtins and Method Invocation

ROP_INVOKE_MUT mirrors the stack VM's OP_INVOKE_MUT. Its third word carries skip, a forward count of RegInstr words past the write-back chain; a genuine in-place builtin mutation falls through into the chain, everything else advances ip by skip once and resolves like an ordinary invoke.

B.2.10 Iterators, Phase, Exceptions, and Defer

B.2.11 Advanced Phase, Concurrency, and Modules

ROP_SCOPE is followed by a data word carrying spawn_count in A and sync_idx in B, then the spawn sub-chunk indices packed three per word. ROP_SELECT_CHOOSE takes the pair destination in A, the

Opcode	Fmt	Operands	Semantics
BIT_AND	ABC	A, B, C	R[A] = R[B] & R[C]
BIT_OR	ABC	A, B, C	R[A] = R[B] | R[C]
BIT_XOR	ABC	A, B, C	R[A] = R[B] ^ R[C]
BIT_NOT	ABC	A, B	R[A] = ~R[B]
LSHIFT	ABC	A, B, C	R[A] = R[B] « R[C]
RSHIFT	ABC	A, B, C	R[A] = R[B] » R[C]

Opcode	Fmt	Operands	Semantics
JMP	sBx	sBx(24)	ip += sBx
JMPFALSE	AsBx	A, sBx	If !R[A]: ip += sBx
JMPTRUE	AsBx	A, sBx	If R[A]: ip += sBx
JMPNOTNIL	AsBx	A, sBx	If R[A] != nil: ip += sBx

operand base register in B, and the arm count in C, followed by $\lceil C/3 \rceil$ raw words packing one arm's flags per field; it produces `[selected_index, received_value]`, with index -1 when no arm fires. `ROP_CHECK_MUTABLE` is the guard emitted ahead of an in-place compound assignment into a register, and is what makes `fix y = 1; y += 1` inside a function raise `cannot modify a frozen value` on this backend.

B.2.12 Pattern Matching

B.2.13 Optimization Fast-Path and Halt

B.3 Bytecode Files

Compiled bytecode is written by `clat compile`:

```
clat compile prog.lat -o prog.latc          # stack VM    -> LATC
clat compile --regvm prog.lat -o prog.rlat  # register VM -> RLAT
clat prog.latc                              # run it
```

Each file starts with a four-byte magic (`LATC` or `RLAT`) followed by a little-endian 16-bit format version. Both formats are at **version 3** in v0.6.4, and both accept a minimum of version 2, so files produced by v0.4.1 through v0.6.x still load.

Opcode	Fmt	Operands	Semantics
GETGLOBAL	ABx	A, Bx	R[A] = globals[K[Bx]]
SETGLOBAL	ABx	A, Bx	globals[K[Bx]] = R[A]
DEFINEGLOBAL	ABx	A, Bx	Define globals[K[Bx]] = R[A]
GETFIELD	ABC	A, B, C	R[A] = R[B].field[K[C]]
SETFIELD	ABC	A, B, C	R[A].field[K[B]] = R[C]
GETFIELD_16	2-word	A, B; Bx	Wide field key: R[A] = R[B].field[K[Bx]]
SETFIELD_16	2-word	A, C; Bx	Wide field key: R[A].field[K[Bx]] = R[C]
GETINDEX	ABC	A, B, C	R[A] = R[B][R[C]]
SETINDEX	ABC	A, B, C	R[A][R[B]] = R[C]
SETINDEX_LOCAL	ABC	A, B, C	In-place R[A][R[B]] = R[C]
GETUPVALUE	ABC	A, B	R[A] = Upvalue[B]
SETUPVALUE	ABC	A, B	Upvalue[B] = R[A]
CLOSEUPVALUE	A	A	Close upvalue at R[A]

Opcode	Fmt	Operands	Semantics
CALL	ABC	A, B, C	Call R[A] with args R[A+1..A+B]; results in R[A..A+C-1]
RETURN	ABC	A, B	Return R[A..A+B-1]
CLOSURE	ABx	A, Bx	R[A] = closure(K[Bx])
COLLECT_VARARGS	ABC	A, B	Collect excess args $\geq$ B into array at R[A]

B.3.1 Sub-chunk constants

A chunk's constant pool holds more than literals. Every nested code body — a function, a `spawn` body, a `scope` sync block, a `select` arm, a `defer` cleanup — is stored as a `VAL_CLOSURE` constant with `body == NULL` and a `Chunk *` in the `native_fn` field. The variable-length operands of `OP_SCOPE` and `OP_SELECT` are indices into that pool.

Format v3 is what completed the picture: `match`, `select`, and `defer` all compile to dedicated opcodes and sub-chunks, so the *entire* language now serializes — there are no remaining AST dependencies in either bytecode path. Nullary enum-variant constants survive an `.rlat` round trip as well.

B.3.2 Load-time verification

A bytecode file is untrusted input, and the VM dispatch loops trust every operand they read. So every deserialized chunk is verified before it executes, by `chunk_verify()` in `src/chunk.c` and its register-side counterpart. Verification checks that:

Opcode	Fmt	Operands	Semantics
NEWARRAY	ABC	A, B, C	R[A] = [R[B..B+C-1]]
NEWTUPLE	ABC	A, B, C	R[A] = tuple(R[B..B+C-1])
NEWSTRUCT	ABC	A, B, C	R[A] = struct K[B] with C fields
NEWENUM	ABC	A, B, C	R[A] = enum(base=B, argc=C)
BUILDRANGE	ABC	A, B, C	R[A] = range(R[B], R[C])
LEN	ABC	A, B	R[A] = len(R[B])
ARRAY_FLATTEN	ABC	A, B	R[A] = flatten(R[B])

Opcode	Fmt	Operands	Semantics
PRINT	ABC	A, B	Print R[A..A+B-1]
INVOKE	2-word	dst, method key (split), argc, obj, args base	Invoke a method on a register
INVOKE_GLOBAL	2-word	dst, name, argc, method, base	Invoke on a global; write back
INVOKE_LOCAL	2-word	dst, reg, argc, method, base	Invoke on a local; mutate in place
INVOKE_UPVALUE	2-word	dst, uv_idx, argc, method, base	Invoke through an upvalue cell
INVOKE_MUT	3-word	dst, method, argc, obj, base, skip	Nested-element mutating call plus write-back chain
FREEZE	ABC	A, B	R[A] = freeze(R[B])
THAW	ABC	A, B	R[A] = thaw(R[B])
CLONE	ABC	A, B	R[A] = clone(R[B])

- every opcode is known and none of its operands run past the end of the code — including the count-driven variable parts of OP_SCOPE, OP_SELECT, OP_SELECT_CHOOSE, and the two CLOSURE forms;
- every constant index is in range *and of the expected type* — a name operand must point at a string, a sub-chunk operand at a closure;
- every jump target, and every OP_INVOKE_MUT skip target, lands on an instruction boundary;
- the code ends in a control-terminating instruction (OP_HALT, OP_RETURN, or OP_THROW);
- sub-chunk recursion stays within 200 levels.

Failures are ordinary load errors. Lopping the tail off a working `.latc` gives `error: truncated: incomplete bytecode`; cutting it down below the header gives `error: file too small for .latc header`.

Opcode	Fmt	Operands	Semantics
`ITERINIT`	ABC	A, B	Init iterator from `R[B]`; index at `R[A+1]`
`ITERNEXT`	ABC	A, B, sBx	Next value or jump
`MARKFLUID`	A	A	Tag `R[A]` fluid (privatizing a crystal)
`SUBLIMATE`	A	A	`R[A].phase = SUBLIMATED`
`PUSH_HANDLER`	AsBx	A, sBx	Push handler; catch at `ip+sBx`
`POP_HANDLER`	—	—	Pop handler on success
`THROW`	A	A	Throw `R[A]`
`TRY_UNWRAP`	A	A	Unwrap ok or propagate error
`DEFER_PUSH`	ABC	A, B	Register the closure in `R[A]` at scope depth B
`DEFER_RUN`	ABC	A	Run defers with `scope_depth` $\geq$ `A`, LIFO

Verification is structural, not a checksum

These checks prove a file is *well-formed*, not that it is the file you compiled. There is no content hash: a benign byte flip inside constant data can pass verification and change what the program prints. Treat a `.latc` file the way you would treat any other build artifact — verify it in transit by other means if that matters to you.

Opcode	Fmt	Operands	Semantics
FREEZE_VAR	ABC	A, B, C	Freeze var K[A], location B:C
THAW_VAR	ABC	A, B, C	Thaw var K[A]
SUBLIMATE_VAR	ABC	A, B, C	Sublimate var K[A]
REACT	ABC	A, B	Register reaction R[A] on R[B]
UNREACT	A	A	Remove reaction R[A]
BOND	ABC	A, B	Bond R[A] depends on R[B]
UNBOND	A	A	Remove bond R[A]
SEED	ABC	A, B	Register seed contract
UNSEED	A	A	Remove seed
IS_CRYSTAL	ABC	A, B	R[A] = (R[B].phase == CRYSTAL)
IS_FLUID	ABC	A, B	R[A] = (R[B].phase == FLUID)
FREEZE_FIELD	ABC	A, B, C	Freeze R[A].field[K[B]]
THAW_FIELD	ABC	A, B, C	Thaw R[A].field[K[B]]
FREEZE_EXCEPT	2-word	A, B, C, base, count	Freeze all except the listed keys
CHECK_TYPE	2-word	A, Bx; err	Throw if type(R[A]) != K[Bx]
CHECK_MUTABLE	A	A	Throw if R[A] is crystal or sublimated
CHECK_ENUM_VARIANT	2-word	A, B; Bx	Validate imported enum metadata and payload arity
IMPORT	ABx	A, Bx	R[A] = import(K[Bx])
REQUIRE	ABx	A, Bx	R[A] = require(K[Bx])
SCOPE	var	—	Spawn concurrent tasks (see below)
SELECT	var	—	Legacy channel select
SELECT_CHOOSE	var	A, B, C	Choose among pre-evaluated arm operands
SETSLICE	ABC	A, B, C	Splice R[A][R[B]..] = R[C]
SETSLICE_LOCAL	ABC	A, B, C	In-place splice
RESET_EPHEMERAL	—	—	Reset bump arena (opcode 80)

Opcode	Fmt	Operands	Semantics
MATCH_TYPE	ABC	A, B, C	R[A] = (tag(R[B]) == ValueType C)
MATCH_FIELD	2-word	A, B, C; Bx	R[A] = whether R[C] has field K[Bx]; value into R[B]
MATCH_PHASE	ABC	A, B, C	R[A] = (phase(R[B]) == AstPhase C)

Opcode	Fmt	Operands	Semantics
`ADD_INT`	ABC	A, B, C	`R[A] = R[B].int + R[C].int` (no check)
`SUB_INT`	ABC	A, B, C	`R[A] = R[B].int - R[C].int`
`MUL_INT`	ABC	A, B, C	`R[A] = R[B].int * R[C].int`
`LT_INT`	ABC	A, B, C	`R[A] = R[B].int < R[C].int`
`LTEQ_INT`	ABC	A, B, C	`R[A] = R[B].int <= R[C].int`
`INC_REG`	A	A	`R[A]++` (assumes int)
`DEC_REG`	A	A	`R[A]--` (assumes int)
`HALT`	—	—	Stop execution (opcode 100)

Backend	Magic	Current	Minimum accepted
Stack VM	`LATC`	3	2
Register VM	`RLAT`	3	2

Table B.3: Bytecode format versions (`include/latc.h`).

Appendix C

Phase Quick Reference

Lattice's *phase system* gives every runtime value a mutability state. This appendix collects all the rules in one place for quick look-up.

C.1 Phase Tags at a Glance

Every `LatValue` carries a `PhaseTag` field:

Tag	Symbol	Keyword	Meaning
FLUID	`~`	`flux`	Mutable. May be reassigned and modified in place.
CRYSTAL	`*`	`fix`	Immutable. Any mutation attempt is a runtime error.
UNPHASED	—	`let`	Default. Treated as fluid in casual mode.
SUBLIMATED	—	`sublimate`	Sealed in place: no mutation, and no channel sends. Not irreversible—`thaw()` still yields a fluid copy.

Table C.1: Runtime phase tags.

Keyword	Phase	Notes
`let`	UNPHASED	Default binding. In casual mode, behaves as fluid. In strict mode, produces a phase-checker error—use `flux`/`fix` instead.
`flux`	FLUID	Explicitly mutable binding. Binding a *crystal* value with `flux` privatizes the value and melts the copy—an implicit `thaw`. The original binding stays crystal and unmodified.
`fix`	CRYSTAL	Explicitly immutable binding. `fix` hardens the *value*, not the name: in casual mode the name may still be pointed at something else, and the newcomer arrives with its own phase. Only strict mode makes assignment to a `fix` binding a phase error.

Table C.2: Variable declaration keywords and their phases.

C.2 Binding Keywords

The Write-Back Rule

Hand `freeze()`, `thaw()`, or `sublimate()` a *bare variable name* and the new phase is written back into that binding:

```
fix frozen_list = freeze([10, 20, 30])
flux thawed = thaw(frozen_list)

print(phase_of(frozen_list))  // "fluid"  -- melted in place
print(thawed)                 // [10, 20, 30]  -- a separate copy
```

Any other expression shape is left alone, so `freeze(clone(x))` hardens only the copy. Identical on all three backends. See Sections II.5.2 and II.6.4.

C.3 Phase Transition Operations

Operation	Input	Output	Notes
`freeze(x)`	any	CRYSTAL	Deep-freezes; returns a crystal clone. On a bare name, the crystal is written back into that binding too. Since v0.5.0, immutability enforced on all backends. Since v0.6.0, frozen containers are shared by ref for O(1) copying.
`freeze(x) where` *fn*	any	CRYSTAL	Freeze with contract validation. The contract must *raise* to refuse; a returned `false` does not stop the freeze.
`freeze(x) except [...]`	struct/map	CRYSTAL (map), per-field FLUID (struct)	Freeze all keys/fields except listed ones. Since v0.6.4, an exempt key is writable by index/field assignment *and* by the single-key methods `set`/`remove`; whole-container mutators such as `merge` still raise. Apply it to a bare name—on an rvalue the exemptions are lost. A later full `freeze(x)` closes exemptions.
`thaw(x)`	crystal	FLUID	Returns a mutable clone. Accepts a SUBLIMATED value as well. Since v0.6.0, always a fresh copy—never affects other aliases. On a bare name, that binding is melted too. Error if already fluid in strict mode.
`clone(x)`	any	same as input	Deep-clones preserving the current phase. On shared frozen values, creates a guaranteed physical copy to detach from the shared structure.

Operation	Input	Output	Notes
`sublimate(x)`	any	SUBLIMATED	Blocks in-place mutation and channel sends—the strongest *in-place* guarantee, not a vault. On a bare name the phase is written back into the binding. `thaw()` still returns a fluid copy (and melts the name); `thaw(clone(x))` leaves the seal on `x` intact.

Table C.3: Phase transition operations.

C.4 Advanced Phase Constructs

Construct	Description
`anneal(x) |v| { . . . }`	Temporarily thaw a crystal value into `v`, execute the block, then re-freeze. The result is crystal.
`forge { . . . }`	Execute a mutable construction block. All mutations inside are allowed; the final result is frozen upon exit.
`crystallize(x) { . . . }`	Deep-freeze `x`, then execute a block with the frozen value in scope. `x`'s original phase is restored when the block ends.
`borrow(x) { . . . }`	The inverse: melt a crystal variable for the block's duration so it can be mutated, then re-freeze it. `phase_of(x)` reads CRYSTAL again outside the block, and the mutations stick.

Table C.4: Compound phase constructs.

C.5 Type Annotations with Phases

Phase constraints can appear in type positions.

Composite constraints use a bitmask internally (`PhaseConstraint`):

```
PCON_FLUID = 1    PCON_CRYSTAL = 2    PCON_SUBLIMATED = 4
```

Syntax	Meaning
`~T` or `flux T`	Parameter must be fluid.
`*T` or `fix T`	Parameter must be crystal.
`(~|*) T`	Accepts fluid or crystal (composite constraint).
`(~|*|sublimated) T`	Accepts any explicit phase.
`T`	No constraint (unspecified).

Table C.5: Phase-annotated type syntax.

C.6 Reactive Phase System

Lattice provides a reactive layer on top of the phase system. These operations register callbacks and dependencies that fire when a variable's phase changes.

Keyword	Description
`react(name, callback)`	Register a reaction callback on the named variable. Fires when the variable is frozen, thawed, or sublimated.
`unreact(name)`	Remove a previously registered reaction.
`bond(target, dep, strategy)`	Create a dependency bond: when `dep` changes phase, `target` is updated according to `strategy` (`"freeze"`, `"thaw"`, etc.).
`unbond(target, dep)`	Remove a dependency bond.
`seed(name, contract)`	Register a seed contract—a predicate that must hold whenever the variable transitions to crystal.
`unseed(name)`	Remove a seed contract.

Table C.6: Reactive phase operations.

C.7 Phase Queries

C.8 Mode Comparison

C.9 Pattern Matching with Phases

Match arms can optionally carry a *phase qualifier*:

Expression	Result
`typeof(x)`	Returns a string: `"Int"`, `"Float"`, `"String"`, etc.
`phase_of(x)`	Returns a string: `"fluid"`, `"crystal"`, `"unphased"`, or `"sublimated"`.
`is_crystal`	Opcode-level check; pushes `true` if CRYSTAL.
`is_fluid`	Opcode-level check; pushes `true` if FLUID.

Table C.7: Phase query operations.

```
match x {
    fluid val  => { /* matches only if x is fluid */ }
    crystal val => { /* matches only if x is crystal */ }
    val        => { /* matches regardless of phase */ }
}
```

The qualifier is checked against the scrutinee's runtime phase tag *before* the structural pattern is tested.

C.10 Per-Field Phase Control

Struct values support per-field phase tracking. When `field_phases` is non-`NULL`, each field can independently be fluid or crystal:

```
struct Config { host: String, port: Int }

flux config = Config { host: "localhost", port: 8080 }
freeze(config) except ["port"]   // port stays fluid, host frozen

print(phase_of(config))  // "fluid" --- the struct carries per-field phases
config.port = 9090       // OK
// config.host = "x"     // error: cannot assign to frozen field 'host'
```

Apply `except` to a *bare name*. Freezing an rvalue and binding the result—`let c = freeze(Config { ... }) except ["port"]`—produces a fully crystal struct with no exempt fields, and the write is refused.

Rule	Casual Mode	Strict Mode
`let` binding	Allowed; phase inherited from initializer	Error: must use `flux` or `fix`
`freeze` on crystal	Allowed; refreezes and closes any `except` holes	Error: "cannot freeze an already crystal value"
`thaw` on fluid	Allowed; still hands back a fresh copy	Error: "cannot thaw an already fluid value"
Assignment to `fix`	Allowed—the name is rebound, and the new value brings its own phase	Phase error: "cannot assign to crystal binding"
`flux` binding from a crystal	Allowed; privatizes and melts the copy	Phase error: "cannot bind crystal value with flux"
`freeze` consumes the binding	No	Yes, but only under `--tree-walk` (see Section 14.1)
`spawn` with fluid	Allowed	Error: "cannot use fluid binding across thread boundary"
Destructure without phase	Allowed	Error: requires explicit `flux`/`fix`
Function arg phase mismatch	Runtime error for the `~T`/`*T` prefix forms; parenthesized constraints are unchecked	Phase error: no matching overload

Table C.8: Casual vs. strict mode phase enforcement.

A struct frozen this way reports FLUID because its per-field table is what carries the phases; a *map* frozen with `except` reports CRYSTAL instead, with the exemptions recorded alongside it.

The `OP_FREEZE_FIELD` opcode freezes a single field, while `OP_FREEZE_EXCEPT` freezes all fields except the listed exceptions.

C.11 Phase Dispatch (Overloading)

Functions with the same name but different phase-annotated parameter types form an *overload set*. In strict mode, the phase checker verifies that at least one overload matches the caller's argument phases:

```
fn process(data: ~Array) -> String { /* fluid path */ }
fn process(data: *Array) -> String { /* crystal path */ }
```

The checker walks the overload chain via `next_overload` pointers, comparing each candidate's parameter phases against the call-site argument phases.

In casual mode there is no checker, but the `~T`/`*T` prefix forms (and their `flux T`/`fix T` spellings) are still enforced at call time. When no overload accepts the argument's phase, the call is refused before the body runs—and the wording depends on the backend:

Backend	Message
stack VM, `--regvm`	`phase constraint violation in function 'mutate'`
`--tree-walk`	`function 'mutate' parameter 'data' requires flux argument, got crystal (fix)`

Table C.9: Phase-constraint refusal, by backend.

The parenthesized composite forms (`(~|*) T` and friends) are a strict-mode-only contract: the static checker tests them, and nothing checks them at runtime.

Appendix D

Built-in Functions Reference

This appendix is an alphabetical listing of every built-in function and method in the Lattice standard library. Global functions are listed first, followed by methods grouped by receiver type.

D.1 Global Functions

Table D.1: Global built-in functions.

Function	Signature	Description
`chr`	`(code: Int) -> String`	Return the character for ASCII code `code`.
`clone`	`(val: Any) -> Any`	Deep-clone `val`, preserving its current phase. Since v0.6.0, `clone()` on a shared frozen value creates a guaranteed physical copy—use it to detach a small piece from a large shared structure.
`freeze`	`(val: Any) -> Any`	Deep-freeze `val`; returns a crystal copy. When `val` is a bare variable name, the crystal is also written back into that binding—use `freeze(clone(x))` to harden only the copy. Since v0.5.0, immutability is runtime-enforced on every backend (index/field assignment and mutating methods all raise errors). Since v0.6.0, frozen containers may be stored in shared immutable storage for O(1) copying.
`input`	`(prompt: String?) -> String?`	Read a line from stdin. Returns `nil` on EOF.
`len`	`(val: Any) -> Int`	Return the length of an array, string, map, set, buffer, or tuple.

Function	Signature	Description
`ord`	`(s: String) -> Int`	Return the ASCII value of the first character. −1 for empty string.
`parse_float`	`(s: String) -> Float?`	Parse a string to a float. Returns `nil` on failure.
`parse_int`	`(s: String) -> Int?`	Parse a string to an integer. Returns `nil` on failure.
`phase_of`	`(val: Any) -> String`	Return the phase as a string: `"fluid"`, `"crystal"`, `"unphased"`, or `"sublimated"`.
`print`	`(...args: Any) -> Unit`	Print values to stdout, space-separated, with trailing newline.
`read_file`	`(path: String) -> String?`	Read entire file. Returns `nil` on error.
`sublimate`	`(val: Any) -> Any`	Permanently freeze `val`: no in-place mutation, no channel sends. On a bare variable name the new phase is written back into the binding. Not a vault—`thaw()` still hands back a fluid copy.
`thaw`	`(val: Any) -> Any`	Thaw a crystal value; returns a fluid copy. Since v0.6.0, always creates a fresh mutable copy—other aliases of the frozen value are unaffected. When the argument is a bare variable name, that binding is melted too.
`to_string`	`(val: Any) -> String`	Convert any value to its string representation.
`typeof`	`(val: Any) -> String`	Return the type name: `"Int"`, `"Float"`, `"Bool"`, `"String"`, `"Array"`, `"Struct"`, `"Closure"`, `"Unit"`, `"Nil"`, `"Range"`, `"Map"`, `"Channel"`, `"Enum"`, `"Set"`, `"Tuple"`, `"Buffer"`, `"Ref"`, `"Iterator"`.
`write_file`	`(path: String, data: String) -> Bool`	Write string to file. Returns `true` on success.

D.2 Array Methods

D.2.1 Non-Closure Methods

D.2.2 Closure Methods

These methods take a closure as their last argument.

Table D.3: Array methods (closure-based).

Method	Signature	Description
`.all`	`(|el| -> Bool) -> Bool`	`true` if predicate holds for every element.

Method	Signature	Description
`.any`	`(|el| -> Bool) -> Bool`	`true` if predicate holds for any element.
`.each`	`(|el| -> Unit) -> Unit`	Call closure for each element (side effects). Not implemented under `--tree-walk`, which raises `unknown method '.each()' on Array`.
`.filter`	`(|el| -> Bool) -> [Any]`	Return elements satisfying predicate.
`.find`	`(|el| -> Bool) -> Any`	First element satisfying the predicate, or **Unit** if none matches—not `nil`.
`.find_index`	`(|el| -> Bool) -> Int`	Index of first match, or -1.
`.flat_map`	`(|el| -> [Any]) -> [Any]`	Map then flatten one level.
`.group_by`	`(|el| -> String) -> Map`	Group elements by key function.
`.map`	`(|el| -> Any) -> [Any]`	Apply closure to each element.
`.partition`	`(|el| -> Bool) -> [[Any]]`	Split into `[matches, rest]`.
`.reduce`	`(|acc, el| -> Any, init: Any) -> Any`	Fold from left. **Closure first, seed second.** Both arguments are required on the default VM and `--tree-walk`; `--regvm` also accepts the closure alone, seeding from the first element. Seed-first evaluates to `nil` on the VMs and raises `.reduce() first argument must be a closure` under `--tree-walk`.
`.sort_by`	`(|a, b| -> Int) -> [Any]`	Sort with a two-argument *comparator* returning negative, zero, or positive—not a one-argument key function. A one-parameter closure raises `expected 1 arguments but got 2`.

D.3 String Methods

D.4 Map Methods

D.5 Set Methods

Method	Signature	Description
`.chunk`	`(n: Int) -> [[Any]]`	Split into sub-arrays of size `n`.
`.contains`	`(val: Any) -> Bool`	Test if array contains `val`.
`.drop`	`(n: Int) -> [Any]`	Return array without the first `n` elements.
`.enumerate`	`() -> [[Int, Any]]`	Return array of `[index, value]` pairs.
`.first`	`() -> Any`	First element, or **Unit** if empty—not `nil`. `== nil` is `false` and `??` does not fire; test with `typeof(x) == "Unit"`.
`.flatten`	`() -> [Any]`	Flatten one level of nesting. Not implemented under `--tree-walk`.
`.index_of`	`(val: Any) -> Int`	Index of first occurrence, or -1.
`.join`	`(sep: String) -> String`	Join elements into a string with separator.
`.last`	`() -> Any`	Last element, or **Unit** if empty (see `.first`).
`.max`	`() -> Int|Float`	Maximum numeric element. *Raises* on an empty array (`max() called on empty array`).
`.min`	`() -> Int|Float`	Minimum numeric element. *Raises* on an empty array.
`.reverse`	`() -> [Any]`	Return reversed copy.
`.sum`	`() -> Int|Float`	Sum all numeric elements.
`.take`	`(n: Int) -> [Any]`	Return first `n` elements.
`.unique`	`() -> [Any]`	Return array with duplicates removed.
`.zip`	`(other: [Any]) -> [[Any]]`	Zip two arrays into pairs.

Table D.2: Array methods (non-closure).

D.6 Buffer Methods

D.7 Enum Methods

D.8 Universal Properties

The following properties are available on *all* collection types via the `len` built-in or the `.length`/subscript pattern. There is no `.length` method—use `len(x)` instead.

Method	Signature	Description
`.bytes`	`() -> [Int]`	Return array of byte values.
`.chars`	`() -> [String]`	Return array of single characters.
`.contains`	`(sub: String) -> Bool`	Test if substring exists.
`.count`	`(sub: String) -> Int`	Count non-overlapping occurrences.
`.ends_with`	`(suffix: String) -> Bool`	Test suffix.
`.index_of`	`(sub: String) -> Int`	First index of substring, or -1.
`.is_alpha`	`() -> Bool`	All characters are alphabetic.
`.is_alphanumeric`	`() -> Bool`	All chars are alphanumeric.
`.is_digit`	`() -> Bool`	All characters are digits.
`.is_empty`	`() -> Bool`	String has zero length.
`.last_index_of`	`(sub: String) -> Int`	Last index of substring, or -1.
`.pad_left`	`(width: Int, ch: String) -> String`	Pad on the left to `width`.
`.pad_right`	`(width: Int, ch: String) -> String`	Pad on the right to `width`.
`.repeat`	`(n: Int) -> String`	Repeat string `n` times.
`.replace`	`(old: String, new: String) -> String`	Replace all occurrences.
`.reverse`	`() -> String`	Reverse the string.
`.split`	`(sep: String) -> [String]`	Split by separator.
`.starts_with`	`(prefix: String) -> Bool`	Test prefix.
`.substring`	`(start: Int, end: Int) -> String`	Extract substring by index range.
`.to_lower`	`() -> String`	Convert to lowercase.
`.to_upper`	`() -> String`	Convert to uppercase.
`.trim`	`() -> String`	Strip whitespace from both ends.
`.trim_end`	`() -> String`	Strip trailing whitespace.
`.trim_start`	`() -> String`	Strip leading whitespace.

Table D.4: String methods.

Method	Signature	Description
`.entries`	`() -> [[String, Any]]`	Return array of `[key, value]` pairs.
`.get`	`(key: String) -> Any`	Value for `key`, or `Nil` if absent. Exactly one argument—there is no default parameter. Write `m.get(k) ?? fallback`.
`.has`	`(key: String) -> Bool`	Test if key exists.
`.keys`	`() -> [String]`	Return array of all keys.
`.merge`	`(other: Map) -> Unit`	Merge `other` into the receiver *in place*; `other` wins on conflicts. Returns Unit, not a new map.
`.remove`	`(key: String) -> Unit`	Remove `key` from the receiver in place. Returns Unit. (`.delete` exists only under `--regvm`.)
`.values`	`() -> [Any]`	Return array of all values.

Table D.5: Map methods.

Method	Signature	Description
`.add`	`(val: Any) -> Unit`	Add element to the receiver in place.
`.clear`	`() -> Unit`	Empty the receiver in place.
`.difference`	`(other: Set) -> Set`	Elements in `self` but not `other`.
`.has`	`(val: Any) -> Bool`	Membership test.
`.intersection`	`(other: Set) -> Set`	Elements in both sets.
`.is_subset`	`(other: Set) -> Bool`	All elements in `other`?
`.is_superset`	`(other: Set) -> Bool`	Contains all of `other`?
`.remove`	`(val: Any) -> Unit`	Remove element from the receiver in place.
`.symmetric_difference`	`(other: Set) -> Set`	XOR of two sets.
`.to_array`	`() -> [Any]`	Convert to array.
`.union`	`(other: Set) -> Set`	Combine both sets.

Table D.6: Set methods.

Method	Signature	Description
`.clear`	`() -> Unit`	Reset the receiver to zero length in place.
`.fill`	`(byte: Int) -> Unit`	Fill the receiver with a byte value in place.
`.push`	`(byte: Int) -> Unit`	Append one byte (u8).
`.push_u16`	`(val: Int) -> Unit`	Append unsigned 16-bit LE.
`.push_u32`	`(val: Int) -> Unit`	Append unsigned 32-bit LE.
`.read_f32`	`(off: Int) -> Float`	Read 32-bit float at offset.
`.read_f64`	`(off: Int) -> Float`	Read 64-bit float at offset.
`.read_i8`	`(off: Int) -> Int`	Read signed 8-bit int.
`.read_i16`	`(off: Int) -> Int`	Read signed 16-bit int.
`.read_i32`	`(off: Int) -> Int`	Read signed 32-bit int.
`.read_i64`	`(off: Int) -> Int`	Read signed 64-bit int.
`.read_u8`	`(off: Int) -> Int`	Read unsigned 8-bit int.
`.read_u16`	`(off: Int) -> Int`	Read unsigned 16-bit int.
`.read_u32`	`(off: Int) -> Int`	Read unsigned 32-bit int.
`.read_u64`	`(off: Int) -> Int`	Read unsigned 64-bit int.
`.resize`	`(size: Int) -> Unit`	Resize the receiver in place (truncate or zero-fill).
`.slice`	`(start: Int, end: Int) -> Buffer`	Return a new sub-buffer; the receiver is untouched.
`.to_array`	`() -> [Int]`	Convert bytes to int array.
`.to_hex`	`() -> String`	Hex-encoded string.
`.to_string`	`() -> String`	Interpret bytes as UTF-8 string.
`.write_i64`	`(off: Int, val: Int) -> Unit`	Write signed 64-bit int.
`.write_u8`	`(off: Int, val: Int) -> Unit`	Write unsigned 8-bit int.
`.write_u16`	`(off: Int, val: Int) -> Unit`	Write unsigned 16-bit int.
`.write_u32`	`(off: Int, val: Int) -> Unit`	Write unsigned 32-bit int.
`.write_u64`	`(off: Int, val: Int) -> Unit`	Write unsigned 64-bit int.

Table D.7: Buffer methods.

Method	Signature	Description
`.enum_name`	`() -> String`	Return the enum type name (e.g. `"Option"`).
`.is_variant`	`(name: String) -> Bool`	Test if variant matches `name`.
`.payload`	`() -> [Any]`	Payload values as an *array*, always—`[1.5]` for a one-field variant, `[]` for a payload-less one. Never `nil`.
`.tag`	`() -> String`	Return the variant name (e.g. `"Some"`).

Table D.8: Enum methods.

Type	`len(x)` returns
`Array`	Number of elements
`String`	Byte length
`Map`	Number of key-value pairs
`Set`	Number of elements
`Tuple`	Number of elements
`Buffer`	Number of bytes
`Range`	`end - start` (may be negative)

Table D.9: Semantics of `len()` by type.

Index

* (fix shorthand), 65
+=, 90
–break flag, 807
–debug flag, 805, 806
–gc-stress, 805
–regvm
    ignored for bytecode, 877
–stats, 8
–stats flag, 805, 837
.., 56
.. (range operator), 98
... (rest pattern), 205
... (spread), 156
... (spread/variadic), 121
.dylib, 891
.env file, 939
.latc file format, 865
    v3, 867
.latc files, 865
.rlat file format, 865
.rlat files, 865
.so, 891
=> prefix (REPL), 12, 25
? operator, 242
    mechanics, 243
?., 55
??, 55
??, 101
@crystal, 351
@fluid, 351
#mode, 71
#mode strict, 17
#mode strict, 345
_ (wildcard), 199
~/.lattice/packages, 745
~ (flux shorthand), 65
405 Method Not Allowed, 965

abs, 641
.all(), 153
alloys, 317–344
    definition, 319
    design patterns, 322
    struct declarations, 317
anneal, 277
annotation
    @crystal, 996
    @fluid, 996
anonymous functions, 123
.any(), 153
API
    JSON, 981
append_file, 549
arena allocation, 300
    definition, 300
args, 634
args(), 13
argument parsing, 933
Array, 143
array
    closure methods, 149
    destructuring, 204

indexing, 144
methods, 147
nested mutation, 146
slicing, 144
arrays, 143
assert, 785
`assert()`, 252
assert_contains, 789
assert_eq, 785, 786
`assert_eq()`, 252
assert_false, 788
assert_gt, 795
assert_lt, 795
assert_ne, 788
assert_near, 795
assert_nil, 795
assert_throws, 790
assert_true, 788
assert_type, 785, 789
assertions, 785
async iterators
vs channels, 525
`async_filter`, 521
`async_iter`, 521
backend support, 522
definition, 523
`async_map`, 521

backend divergence, 601, 655
backend divergences, 858
backends, 845
async iterator support, 522
divergence, 88, 102
spawn error propagation, 542
Ballistics Lab, 641
base64_decode, 629, 631
base64_encode, 629, 631
binary data, 562
binary-safe strings, 182
block expression, 89
`bond()`, 326–330
dependency, 328
one-shot, 330
bond strategies, 330–332
gate, 332
inverse, 331
mirror, 331
Bool, 39
booleans, 39
bootstrapping, 886
fixed point, 887
`borrow`, 277–278
breadth-first search, 747
break, 94
breakpoint(), 32
breakpoint() function, 822
breakpoints, 814
conditional, 815
function, 814
line, 814
management, 816
broadcast pattern, 537
Buffer, 562
clear, 568
fill, 568
indexing, 564
push, 565
read_f32, 565
read_f64, 565
read_i16, 565
read_i32, 565
read_i64, 565
read_i8, 565
read_u16, 565
read_u32, 565
read_u64, 565
read_u8, 565
resize, 568
slice, 568
to_array, 568

to_hex, 568
to_string, 568
write_i64, 565
write_u64, 565
write_u8, 565
built-in functions, 5
bump arena, 1015
BumpArena, 306–307
bytecode, 848
AST-free, 867
file format, 865, 1025
format versions, 867
integrity, 881
opcode numbering, 1013
verification, 879, 1026
bytecode compilation, 11
bytecode serialization, 865
`.bytes()`, 179

C11 compiler, 5
callbacks, 132
errors, 133
`.camel_case()`, 177
cancellation
with Ref, 490
case transformations, 177
casual mode, 71
rules, 71
ceil, 641
chained calls, 696
`Channel`, 497
close, 503
closed, 499
send error, 504
closing best practice, 504
creation, 497
definition, 498
recv, 501
send, 498
send eligibility, 481, 499
channel multiplexing, 508
`Channel::new()`, 497
channels, 497
vs Ref, 531
`.chars()`, 179
chmod, 551, 556
choosing a backend, 857
`chr()`, 186
chunk_verify, 879
circular dependencies, 739, 747
detection, 748
clamp, 641
clat, 6, 733
package manager, 733
clat add, 736
–git, 737
clat compile, 875, 957
clat doc, 759, 768
HTML, 773
JSON, 772
Markdown, 772
output directory, 771
output formats, 772
plain text, 772
clat fmt, 759
check mode, 761
idempotence, 762
stdin, 762
width, 763
clat init, 733
clat install, 736, 738
internals, 751
clat remove, 736, 739
clat test, 777, 781
discovery, 782
filter, 783
summary, 784
verbose, 784
clat-run, 11, 879, 957
`.clear()`, 162

CLI debugger, 806
cli library, 933
  v0.6.4 parse failure, 934
CLI tool, 933
clone, 70
`clone()`, 269–275
clone(), 390
`close()`, 503
closure, 619
  as struct field, 378
  capture, 126, 128
closures, 5, 113, 123
  accumulators, 693
  captured accumulators, 664
  error propagation, 154
  first example, 16
  parameter syntax, 1004
  stack VM implementation, 852
collections, 143
combined flags, 939
command injection, 750
command-line application, 933
command-line arguments, 13
comments, 56
  block, 57
  doc, 57
  line, 56
  nested, 57
compile error
  too many constants, 875
compiler/latc.lat, 881
compose(), 683
  usage, 685
`compose()`, 135
compound assignment, 90, 160
computed goto, 851
concurrency, 5
  first example, 15
  testing, 540
  thread safety, 493
concurrency patterns, 529
  broadcast, 537
  fan-out/fan-in, 487
  pipeline, 518
  producer-consumer, 505
  request-reply, 538
  worker pool, 535
configuration, 592
constructors, 22
`.contains()`, 176
continue, 94
  compilation, 96
contracts, 137, 250, 997
control flow, 85
cookies, 969
copy-on-write, 274
copy_file, 551, 555
cos, 641
`.count()`, 176
CRLF injection, 604
CRUD operations, 979
crypto module, 623
crystal phase, 4, 63, 64, 265
crystal sharing, 285
  channel sends, 501
  memory model, 304–305
`crystallize`, 277–278
crystallized map, 305
CrystalRegion, 300–303
CSV, 573
csv_parse, 583
csv_stringify, 583
currying, 695
cwd, 634

DAP, 805, 826
data formats, 573
data sharing
  between spawns, 479
database, 977

datetime, 623
datetime_add_duration, 627
datetime_format, 627
datetime_from_epoch, 627
datetime_from_iso, 627
datetime_now, 627
datetime_sub, 627
datetime_to_epoch, 627
datetime_to_iso, 627
datetime_to_local, 627
datetime_to_utc, 627
deadlock, 533
  avoidance, 533
Debug Adapter Protocol, 826
debugger, 805
  backtrace, 813
  call stack, 813
  continue, 820
  expression evaluation, 810
  list command, 808
  variable inspection, 809
  VS Code, 826
  watch, 821
debugging, 805
decrement operators, 54
deep cloning
  in spawns, 479
default parameters, 120
defer, 246
  compilation, 250
  execution order, 247
  return value preservation, 249
  scope, 248
  stack VM, 851
  with errors, 248
delete_file, 551, 555
denial of service, 587
dependencies, 735
  git, 737
dependency graph, 739, 747
dependency resolution, 741
deployment
  CLI tools, 957
depth-first search, 748
describe, 792
design by contract, 137
dispatch loop
  stack VM, 851
divergences
  by backend, 858
doc comments, 57, 765
  formatting, 762
  module-level, 766
doc generator, 759
documentation generation, 768
dotenv, 595
  load_file, 940
  load_opts, 941
  parsing rules, 941
dotenv library, 939
dual-heap architecture, 297
DualHeap, 311
`DualHeap`
  per-thread, 493
duration, 627
duration_from_seconds, 627
duration_to_string, 627

`.each()`, 153
ECS, 392
else, 85
else if, 88
`.ends_with()`, 176
ensure, 137, 138, 997
ensure contract, 250
enum, 397
  construction, 399
  definition, 397
  exhaustiveness, 417
  generic, 454

matching, 403
Option pattern, 414
pattern matching, 208
payload, 402
recursive, 420
Result pattern, 414
state machine, 409
tag, 402
unit variant, 210
variant, 397
enums
in REPL, 25
payload destructuring, 1007
env, 634
env_keys, 634
env_set, 634
environment capture, 126
environment variables, 939
ephemeral allocation, 306
eq (custom equality), 382
`err()`, 232
error handling, 229
in callbacks, 671
error map, 237
error propagation, 133, 154, 242
`error()`, 230
escape sequences, 41, 171, 172
evaluator, 845
examples, 10
exception handling
stack VM, 851
exec, 634
exec_argv, 638
`exec_argv`, 955
execution backends, 845
exhaustiveness checking, 212
boolean, 214
enum, 213
export, 711
constructor pattern, 713
enums, 712
explicit, 711
filtering, 713
keyword, 711
legacy mode, 711
modes, 712
re-exporting, 714
structs, 712
expression
if/else, 86
expression continuation, 116
extension
building, 895
calling convention, 892
search order change, 905
search paths, 904
security, 906
value accessors, 893
value constructors, 893
extension API, 891
extensions, 891
available, 898

facade module, 725
alias limitation, 727
false, 39
fan-out/fan-in, 487
ffi extension, 899
type signatures, 899
file I/O, 549
file_exists, 551
file_size, 551, 554
filesystem, 549
`.filter()`, 149
`.find()`, 153
`.find_index()`, 153
`findi` (example application), 949
first-class functions, 114
fix, 63
`fix`, 265–267

rebinding, 266
flags
CLI, 935
flat installation, 752
`.flat_map()`, 153
Float, 38
floats, 38
floor, 641
fluid phase, 4, 62, 63, 263
FluidHeap, 297–300
flux, 10, 62, 90
`flux`, 263–264
fn keyword, 113
fn standard library, 672, 688
apply_n, 689
chunk, 692
comp, 689
constant, 689
curry, 695
flatten, 692
flip, 689
frequencies, 690, 691
group_by, 690
partial, 696
partition, 690, 691
Result type, 692
sequences, 673
thread, 690
uniq_by, 691
Fn type, 48, 999
Fn type annotation, 118
for, 89
for loop, 92
forge, 14, 77
`forge`, 276–277
`format`, 923
formatter, 759
freeze, 4, 68
deep, 68
except clause, 1005
value vs. binding, 64
where clause, 1005
`freeze()`, 269–275
before spawning, 480
consumes binding, 347
except, 280
memory mechanics, 307
partial freeze, 320
when to use, 282
write-back, 1032
freeze-except, 280–282
from, 709
function
generic, 448
function definition, 113
function overloading
phase-dependent, 356–359
function type, 458
function types, 48
functional programming, 683
best practices, 698
functions, 113
introduction, 9

garbage collection
-gc flag, 311
concurrent, 545
flags, 311–313, 837
incremental mode, 312
mark phase, 298
mark-sweep, 297
stress mode, 312
sweep phase, 298
gc-stress, 802
gcd, 646
generics, 49, 447
enums, 454
functions, 448
non-functional, 998
patterns, 461

roadmap, 466
structs, 451
git dependencies, 737
glob, 551, 554
globals
element access, 1017
`.group_by()`, 151
`grow()`, 337
guard (match), 202

header injection, 604
Hello, World, 8
help flag, 937
higher-order functions, 132
type annotations, 118
history(), 830, 831
hmac_sha256, 629, 630
Homebrew, 5
hostname, 634
HTML templating, 971
HTTP, 597
HTTP headers
case sensitivity, 968
HTTP server, 961
http_get, 597
http_post, 597
http_request, 597
http_server, 610
response helpers, 963
http_server library, 612, 961

identity(), 683
as a value, 688
usage, 687
if expression, 85
image extension, 902
impl, 375, 427
block, 429
multiple blocks, 433
implicit return, 9
import, 1000
aliased, 708
nested limitation, 727
avoiding collisions, 728
bare, 710
tree-walk divergence, 710
basic, 707
caching, 721
in pre-compiled bytecode, 878
internals, 729
name collisions, 710
relative paths, 719
scoped, 728
selective, 709
styles, 708
working directory, 717
imports, 707
increment operator
absent, 1002
increment operator (absent), 90
increment operators, 54
`.index_of()`, 180
infinity, 646
inline cache, 855
installation, 5
Homebrew, 5
Linux, 6
macOS, 6
WebAssembly, 7
Windows, 7
instruction encoding
register VM, 853
Int, 37
integer overflow, 646
integers, 37
integrity verification, 743
interface, 435
is_alpha(), 44
is_alphanumeric(), 44
`is_complete`, 913
is_digit(), 44

is_dir, 551
`is_err()`, 233
`is_error()`, 230
is_file, 551
is_inf, 646
is_nan, 646
`is_ok()`, 233
it, 792
iter(), 659
  supported types, 660
iter_next(), 659
iteration, 92
Iterator type, 659
iterators, 5, 16, 659
  all, 669
  any, 669
  async, 521
  backend differences, 679
  chaining, 667
  channels, 661
  collect, 668
  consumers, 668
  count, 670
  custom, 672
  enumerate, 666
  errors, 671
  filter, 664
  fold, 668
  for loops, 661
  internals, 680
  map, 663
  performance, 677
  protocol, 659
  range, 662
  reduce, 668
  repeat, 662
  skip, 665
  sources, 661
  take, 664
  tap, 664
  transformers, 663
  zip, 666

joining semantics, 476
JSON, 573, 981
  strict parsing, 982
json_parse, 573, 981
json_stringify, 573, 981

`.kebab_case()`, 177

lambda, 123
  no type annotations, 124, 459
last_index_of(), 44
`lat_eval`, 913
  dangers, 925
lat_ext_init, 891
lat_ext_register, 891
lat_modules, 717, 733, 739
  structure, 746
`.latc` file, 1025
LatExtValue, 893
Lattice
  design philosophy, 3
  introduction, 3
  unique features, 14
lattice.lock, 739, 742
  checksum, 743
  format, 742
  package entry, 742
lattice.toml, 723, 733, 734
  creating, 733
  dependencies, 735
  entry field, 719
  package section, 735
  validation, 749
lattice_ext.h, 891
LATTICE_EXT_ALLOW_CWD, 904
LATTICE_EXT_PATH, 904
LATTICE_INSECURE_REGISTRY, 745
LATTICE_REGISTRY, 739, 744

lazy evaluation, 659
  benefits, 677
lazy sequences, 673
lcm, 646
`.len()`, 180
lerp, 641
let, 61
`let`, 267–269
lexer
  from Lattice code, 914
lib/test.lat, 791
libedit, 6
line width, 763
list_dir, 551
literals
  float, 1009
  integer, 1009
lock file, 742
log, 641
  context, 947
  new, 946
log levels, 946
log library, 943
  backend limitation, 943
logging, 943
loop, 89
  infinite, 91
loops, 89

main.lat, 723
Makefile
  extension, 895
manifest, 734
Map, 157
  compound assignment, 160
  methods, 158
  missing key, 102
  nested mutation, 955
map literal, 602
`.map()`, 149
maps, 157
  closures as values, 701
match
  enum patterns, 403
  exhaustiveness, 417
match expression, 195
  arm body, 197
  compilation, 219
  evaluation order, 197
  in operand position, 221
  syntax, 196
materials science, 75
math module, 623, 641
  E, 716
  PI, 716
max, 641
md5, 629
memory architecture, 297–316
memory statistics, 837
metaprogramming, 909–929
  best practices, 925
middleware, 966
  signature, 967
min, 641
mkdir, 551
module
  definition, 711
module caching, 721
module map
  mutation, 723
module resolution, 716
  built-in, 717
  lat_modules, 718
  order, 716
  packages, 717
  path confinement, 719
  relative, 717
  working directory, 717
modules, 707
  built-in, 714

top-level method calls, 943
multi-file projects, 723
multi-line strings, 43
multiline strings, 173
mutability, 261
as first-class concept, 66

namespacing, 728
NaN, 646
native extensions, 891
search order, 719
nested mutation, 146, 1018
nesting depth, 587
networking, 597
Nil, 45
nil, 28, 45, 88
`nil`
vs Unit, 503
nil coalescing, 45
nil-coalescing operator, 101
NUL byte, 182, 599, 606, 641
number literals, 49
hexadecimal, 50
underscores, 50

`ok()`, 232
OP_BUILD_ENUM, 424
OP_CHECK_RETURN_TYPE, 118
OP_CHECK_TYPE, 117
OP_CONSTANT_16, 874
OP_DEFER_PUSH, 867
OP_FREEZE, 65
OP_IMPORT, 729
operand width, 730
OP_INDEX_GLOBAL, 1017
OP_INVOKE_MUT, 694, 1018
OP_INVOKE_UPVALUE, 128
OP_MARK_FLUID, 63
OP_MATCH_TYPE, 867
OP_RESET_EPHEMERAL, 1015
OP_SCOPE, 1021
`OP_SCOPE`, 492
OP_SELECT, 1021
`OP_SELECT`, 525
OP_SELECT_CHOOSE, 867
OP_TRY_UNWRAP, 243
opcodes
fast path, 1022
stack VM, 849
wide constant, 874
wide variants, 1016
OpenSSL, 6
operators, 51
arithmetic, 51
bitwise, 53
comparison, 51
compound assignment, 54
logical, 52
nil coalescing, 55
optional chaining, 55
range, 56
Option, 414
options
CLI, 935
`ord()`, 186
orm library, 977
backend limitation, 977
os module, 623

package cache, 744
global, 745
package manager, 733
fork+exec, 750
security, 749
`panic()`, 245
parameter types, 117
parse error
chained call, 131
parse errors
chained calls, 696
parser

depth limits, 1011
parser limits, 647
parser strictness, 587
partial application, 695
`.partition()`, 153
pass-by-value, 388
path traversal, 749
path_base, 556
path_dir, 556
path_ext, 556
path_join, 556
pattern matching, 5, 195
array destructuring, 204
binding pattern, 199
enum, 403
enum variant, 208
enum variants, 1007
exhaustiveness, 212
first example, 15
guard, 202
idioms, 222
literal pattern, 198
phase qualifier, 216
range pattern, 200
rest pattern, 205
struct destructuring, 207
wildcard, 199
patterns
accumulate then freeze, 77
configuration objects, 78
fluid working copy, 79
performance
backends, 860
persistent VM, 20
pg extension, 900
phase
and channel sends, 499
definition, 262
in patterns, 216
sublimated, 482
phase annotations, 351–353
phase checker, 73, 348
phase constraints, 353–356
definition, 356
runtime enforcement, 356
phase errors, 288–293
phase inference, 267
phase system, 3, 61, 261–296
chemistry metaphor, 75
concurrency, 76
first example, 14
overview, 4
philosophy, 74
phase-aware API design, 359
`phase_of()`, 293
`phase_of`, 909
function, 911
phase_of(), 28
PhaseConstraint, 353
pid, 634
pipe operator, 104
pipe(), 683
usage, 683
`pipe()`, 104
pipeline pattern, 518
platform, 634
polymorphic inline cache, 855
polymorphism, 438
`pop()`, 145
positional arguments, 935
pow, 641
pre-compilation, 875
pressure, 332–335
`pressurize()`, 332–335
producer-consumer, 505
progress bar, 654
progress module, 654
project structure, 723
`push()`, 145

query parameters, 965

random, 641
random_bytes, 629, 632
random_int, 641
range
    in patterns, 200
range_iter(), 662
ranges, 98
raw strings, 171
`react()`, 326–330
reactive bonds, 317–344
reactive data flows, 338
read_file, 549
read_file_bytes, 562
realpath, 551, 556
rebinding, 64, 266
recursion, 116
`recv()`, 501
redis extension, 901
`.reduce()`, 149
`reduce()`, 122
`Ref`, 529
    API, 529
    copy-on-write, 274
    freezing, 531
    get, 529
    proxying, 530
    race conditions, 482
    set, 529
    vs channels, 531
    with spawns, 482
`Ref::new()`, 529
reference counting, 302
reference implementation, 847
reflection, 909–929
RegChunk, 856
regex, 188, 647
    flags, 191
regex flags, 648
regex limits, 651
regex module, 623
regex_find_all, 647
`regex_find_all()`, 189
regex_match, 647
`regex_match()`, 189
regex_replace, 647
`regex_replace()`, 190
region collection, 302
`region_id`, 313
RegionManager, 303
register compiler, 855
register VM, 853
register windows, 854
register-based bytecode, 853
registry, 739, 741, 744
    default, 744
    HTTP, 744
    HTTPS requirement, 745
    override, 744
regular expressions, 188
rename, 551, 555
repeat_iter(), 662
REPL, 19
    backends, 33
    debugging, 32
    defining enums, 23
    defining functions, 23
    defining structs, 23
    error recovery, 31
    exploring built-ins, 30
    history, 30
    incremental building, 29
    introduction, 11
    launching, 19
    multi-line input, 20
    output conventions, 25
    persistent state, 12, 20
    tab completion, 21
    tips and tricks, 28

.replace(), 176
repr, 26
repr, 909
reproducible builds, 742
request Map, 963
request smuggling, 604
request-reply pattern, 538
require, 137, 138, 997
require contract, 250
require_ext, 719
Result, 414
Result type, 232
Result type (fn module), 692
return, 94
return types, 117
rewind(), 830, 833
rmdir, 551
round, 641
round-tripping, 589
routing
    HTTP, 961
    route registration, 964

scope, 15
scope, 471
    as expression, 473
    basic usage, 471
    implementation, 492
    in loops, 478
    nesting, 477
seed(), 335–338
    definition, 336
select, 508
    binding name, 510
    default arm, 514
    definition, 509
    fairness, 510
        determinism, 511
    implementation, 525
    timeout arm, 514
self
    cloning, 381
self-hosted compiler, 881
    format version, 886
    parity, 882
    ser_buf, 886
self-hosting, 886
semantic versioning, 740
semver, 740
    caret, 740
    constraints, 740
    definition, 740
    tilde, 740
send(), 498
Set, 162
    difference, 162
    intersection, 162
    is_subset, 164
    is_superset, 164
    union, 162
sets, 162
SHA-256, 743
sha256, 629
sha512, 629
shell, 634
shell injection, 638
short-circuit evaluation, 53, 102
sign, 646
sin, 641
sleep, 623
sleep(), 477
slicing, 100
.snake_case(), 177
.sort_by(), 151
spawn, 15
spawn, 471
    capture isolation, 479
    data sharing, 479
    error handling, 475
    error propagation, 542

nesting, 490
`.split()`, 174
spread operator, 156
SQLite, 977
sqlite extension, 900
sqrt, 641
stack compiler, 848
stack traces
from bytecode, 881
stack VM, 848
standard library
fn, 688
standard library modules, 714
`.starts_with()`, 176
stat, 551, 554
state machine, 409
static phase checking, 345–364
step-into, 817
step-out, 817
step-over, 817
stepping, 817
STMT_IMPORT, 729
strict mode, 71, 345–364
consume on freeze, 72, 347
enabling, 345
introduction, 17
no let, 72
rules, 72
spawn checking, 350
static checking, 72
String, 40
case conversion, 177
methods, 174
string concatenation, 51
string formatting, 923
string interning, 309–311
string interpolation, 5, 9, 41, 171
disabling, 42
first example, 16
strings, 40, 171
binary-safe, 182
double-quoted, 40
escape sequences, 1009
methods, 44
single-quoted, 42
triple-quoted, 43
struct, 367
callable fields, 378
definition, 367
destructuring, 207, 387
equality, 382
field access, 370
generic, 451
methods, 375
mutation, 370
nested mutation, 372
pass-by-value, 388
reflection, 384
struct reflection, 917–923
struct_fields, 384
`struct_fields`, 917
struct_from_map, 384
`struct_from_map`, 917
function, 920
struct_name, 384
`struct_name`, 917
struct_to_map, 384
`struct_to_map`, 917
function, 919
structs
in REPL, 24
structured concurrency, 471
definition, 472
joining, 476
motivation, 474
structured logging, 943
`sublimate`, 278–280, 482
and channels, 486
definition, 483
sublimated phase, 278, 482

subprocess, 638, 955
.substring(), 180
supply chain, 749

tab completion, 21
  constructors, 22
  keywords, 22
  methods, 23
tan, 641
TCP, 597
tcp_accept, 605
tcp_close, 605
tcp_connect, 605
tcp_listen, 605
tcp_peer_addr, 605
tcp_read, 605
tcp_set_timeout, 605
tcp_write, 605
tempdir, 560
tempfile, 560
template
  conditionals, 973
  filters, 972
  includes, 975
  inheritance, 975
  loops, 975
  replace filter, 973
  variables, 971
template library, 971
test blocks, 777
  syntax, 777
test standard library, 791
testing, 777
  concurrent code, 540
  GC stress, 802
testing strategies, 797
thaw, 4, 68
  creates copy, 69
thaw(), 269–275
  write-back, 1032
thread safety, 493
time, 623
time(), 477
time module, 623
time travel debugging, 830
time_add, 626
time_day, 625
time_format, 623
time_hour, 625
time_minute, 625
time_month, 625
time_parse, 623
time_second, 625
time_weekday, 625
time_year, 625
timeout
  in select, 516
timezone_offset, 627
.title_case(), 177
TLS, 597
tls_available, 608
tls_close, 608
tls_connect, 608
tls_read, 608
tls_write, 608
to_string, 909
tokenize, 914
TOML, 573, 734
toml_parse, 577
toml_stringify, 577
track(), 830, 831
trait, 427
  declaration, 427
  design patterns, 435
  implementation, 429
  internals, 444
  polymorphism, 438
tree-walk
  import resolution, 718
  struct imports, 713

tree-walk backend
    concurrency, 494
tree-walk interpreter, 845
    eager iterators, 679
.trim(), 175
triple-quoted strings, 173
triple-slash comments, 765
true, 39
truthiness, 39
try
    and spawn, 544
try/catch, 235
    and iterators, 672
    as expression, 237
    catchable errors, 240
try_fn(), 253
Tuple, 164
tuples, 164
type annotation, 456
type annotations, 9, 48, 117
    closures, 124
    function types, 999
    lambda parameters, 1004
type checking
    runtime, 448
type expression, 456
typeof, 909
    definition, 910
typeof(), 28, 46
types, 37

Unicode, 186
Unit, 46, 635
Unit
    from recv, 502
unit, 26, 28, 46, 88
unphased, 268
unwrap(), 233
unwrap_or(), 233
upvalue, 619
upvalues, 126, 852
    container mutation, 128
    mutation, 693
URL encoding, 966
url_decode, 619
url_decode, 966
url_encode, 619
uuid, 633

v0.6.4
    captured container mutation, 128
value history, 805, 830
value stack, 850
values, 37
variables, 61
variadic
    parameters, 121
variadic parameters, 120
version constraints, 740
version(), 30
VS Code, 826

watch expressions, 821
web application
    complete example, 984
web server, 610
web service, 961
WebAssembly, 7
websocket extension, 903
while, 89
worker pool, 535
write_file, 549
write_file_bytes, 562

YAML, 573
yaml_parse, 580
yaml_stringify, 580